Next Generation Controller Interface Device

Final Report

April 2011

KLK-209

NIATT Report Number N06-15

Prepared for

OFFICE OF UNIVERSITY RESEARCH AND EDUCATION

U.S. DEPARTMENT OF TRANSPORTATION

Prepared by

National Institute for Advanced Transportation Technology

University of Idaho

Brian Johnson, Richard W. Wall, Michael Kyte, Darcy Bullock, Ron Nelson, Zhen Li, John Fisher, Eugene Bordenkircher, Troy Cuff, Eresh Suwal, Henis Mitro, Sam Young, Zeljko Mijatovic, Tyrel Jensen

Table of Contents

Introduction 1
Part I: Status of CID SDLC Interface 2
Part 2: Traffic Real-time Hardware-in-the-loop Simulation Performance Analysis 4
Introduction 4
Background 5
Hardware-in-the-loop Simulation (HILS) 5
Real-Time Systems 5
HILS Latency Aanalysis 6
Introduction 6
Why Latency Exits 7
HILS Latency Sources 8
Experiments 9
USB 1.1 Communications Experiments 10
CID Hardware Signal Conversion Time Experiments 11
Software Computation Time Experiments 11
Performance Analysis 13
Introduction to Rate Monotonic Scheduling Analysis 14
USB Resource Scheduling Calculation 14
CPU Resource Scheduling Calculation 15
Modeling HILS System Using Petri Nets 16
Latency Analysis Using Queuing Theory 21
Phase Data Package Queue 22
Detector Data Package Queue 22
Summary 23
Conclusion 23
References 24
Part 3: A Method of Verifying the Accuracy of Real-Time Hardware-in-the-Loop Traffic Simulation 26
Chapter 1 – Introduction: Real-Time Hardware-in-the-Loop Traffic Simulation 26

1.1 Traffic Control Systems 26
1.2 Traffic Simulation 26
1.3 Hardware-in-the-Loop Simulation 26
1.4 The Controller Interface Device 27
1.5 Communication Latency and Simulation Accuracy 27
1.6 Verifying the Accuracy of Hardware-in-the-Loop Simulation 28
Chapter 2 – Controller Interface Device Architecture 28
2.1 Overview 28
2.2 Microcontroller 28
2.3 USB Protocol 29
2.4 Traffic Controller Interface 29
2.5 Firmware 29
Chapter 3 – Real-Time Playback 31
3.1 Algorithm Description 31
3.2 Real-Time Playback in Power Systems 32
3.3 Application to Traffic Simulation 32
Chapter 4 – Building a Real-Time Playback Simulator for Traffic Systems 33
4.1 Overview 33
4.2 CID Modifications 33
4.3 Playback Control Program 36
4.4 Simulator Interface 38
Chapter 5 – Real-Time Playback Testing 40
Chapter 6 – Summary & Topics for Future Research 40
References 41
Part 4: Development of a USB for the CID with Reduced Latency 43
Chapter 1 Introduction to the Controller Interface Device (CID II) 43
1.1 Real World Traffic System 43
1.2 Simulating the Real World Traffic 43
1.3 Hardware-in-the-Loop Simulation 43
1.4 Concept of the original controller interface device 44

1.5 Data format for the Controller Interface Device (CID II) 44
1.6 Problems encountered with CID II 45
1.7 Designing a Custom Device Driver 45
1.8 Overview of the Thesis 46
Chapter 2 Introduction to Writing Windows Drivers 47
2.1 Overview of Windows Architecture 47
2.2 Different types of Drivers 47
2.3 Why Write a Windows Driver Model (WDM) Driver? 49
2.4 Data Structures in the Device Drivers 50
2.5 Interrupt Request Level 53
2.6 Summary 54
Chapter 3 Communications using the Universal Serial Bus (USB) 55
3.1 Origin and Classification of USB 55
3.2 USB Bus Topology 55
3.3 Communications in the USB system 55
3.4 USB Transfer Terminology 59
3.5 Types of Transfers 60
Chapter 4 Designing the Controller Interface Device Driver 62
4.1 Control Flow in a Device Driver 62
4.2 Driver Entry Routine 63
4.3 Working with I/O Request Packet (IRP) 64
4.4 Different I/O Request Packets (IRPs) and their Major Codes 66
4.5 The Plug and Play IRPs 66
4.6 Isochronous Transfers 71
4.7 Building, Testing and Installing Device Drivers 74
4.8 Summary 74
Chapter 5 Techniques to Build, Test and Debug a Windows Device Driver 74
5.1 Building Windows Device Drivers 74
5.2 Debugging device drivers 75
5.3 Different Types of Errors 75

5.4 Debugging Routines and Techniques 76
5.5 Debuggers for Windows Operating System 76
Chapter 6 Installing the CID II Windows Driver Model (WDM) Driver 79
6.1 Installing a Windows Device Driver 79
6.2 Sections of a INF file 79
6.3 An Easy Way to Write INF Files 80
Chapter 7 Performance Results from Traffic Measured on the USB Bus 81
7.1 CATC trace of the driver performance 81
7.2 Driver Performance with Multiple Controller Interface Devices 85
Chapter 8 Summary and Future Work 86
8.1 Summary 86
8.2 Suggested Future Work 86
References 88
Appendix A CID II Settings 88
CID Design Team 91

Introduction

Three paths were proposed to develop the future upgrades to the controller interface device (CID). The first was the addition of an RS-485 interface for connecting the CID II to SDLC equipped traffic controllers. The second was to develop a third generation CID capable of 128 inputs and 128 outputs via the traffic controller cables as well as SDLC connection. This will involve several changes to the basic CID layout. The third path involved developing wireless interface between the CID and the traffic controller, allowing more freedom in locating traffic controllers, CIDs and computers in a laboratory facility without the need for long, cumbersome cables. Consideration will be given to the Advanced Transportation Controller and the communications possible with this new controller type. Consideration will also be given to the kind of software tools that need to be developed. Software specifications and prototype development will be conducted.

After studying the project and consulting with our external advisors we realized the following:

1) There is little value is doing a CID that will handle more than128 inputs and outputs. Very few people use that capability in the traffic controller. If they do, the controllers also have a SDLC interface. Higher priority has been given to adding SDLC capability to the existing CID design and developing a SLDC only CID. Initial prototyping for this was performed, but this portion of the project was not completed. Section 1 of this report documents the progress to date. In addition, after conferring with external advisors, there was little interest in a wireless version of the CID, so this part was seen as a low priority
2) The students also spent a fair amount of time working out kinks in the CID II design that was licensed to McCain Traffic supply, slowing progress. The final material was transferred 6 months into the project (the software even later).
3) Instead, there was a lot more interest in verification of the accuracy of the hardware in the loop simulation. This report will present results of both software performance based measures (see abridged summary of the Ph.D. thesis by Zhen Li in Section 2) and hardware based real-time playback system (see MS thesis by John Fisher in Section 3). The real-time playback system was still in a prototype stage at the conclusion of this project.
4) Latency in the USB communications was identified as a key factor limiting the number of traffic controllers that could be interfaced in real-time for a single simulation. The latency arises in the USB driver in running under the operating system. Section 4 of this report documents the development of a new USB driver to improve timing performance (MS Thesis by Manjunatha Reddy-Jayarama).

Part I: Status of CID SDLC Interface

The SDLC implementation is nearly ready for a prototype circuit board to be built. As part of this process, a development board is being designed and built that will support FPGA's, CPLD's, and the current and next generation of the Cypress EZ-USB microcontroller (for prototyping new applications that could use USB 2.0). However, the work on this task was sidetracked to solve the following issues with the CID II design delivered to McCain Traffic supply:

1) Windows XP did not recognize the CID II as a valid USB device. A CID II firmware revision with updated descriptor tables has solved this problem. The revised firmware has been delivered to McCain Traffic Supply.

2) The CID II documentation discusses the capability of utilizing a subset of the pins on NEMA D-connector with the CID, but no detail was provided. The CID NEMA cable was modified, and documentation developed and submitted to McCain traffic supply.

The CID to NEMA traffic controller cable harness can be converted from mode 0 ope ration to include six inputs and six outputs from the NEMA D connector. This conversion is accomplished by using a custom made pig-tail to connect the NEMA D connector to the R and P connectors on the CID/NEMA cable harness (see Figure 1). Since NEMA D connectors vary between manufacturers and applications, a conversion cable is not included.

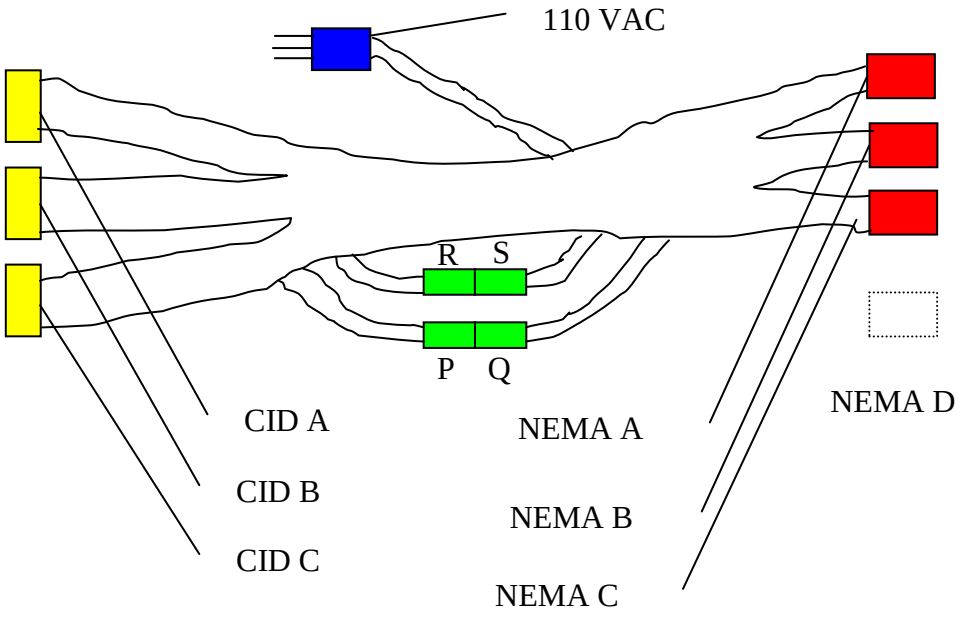

Figure 1 – CID/NEMA connector assignments.

A sketch of the required conversion cable is shown in Figure 2. The S and Q connectors are defined in Table 3. Use the traffic controller manufacturer's documentation to determine what connector should be used for your NEMA D connector. The conductors in the conversion cable should be no smaller than 24 gauge stranded copper wire. The length of the conversion cable is not critical as long as it is less than five feet in length.

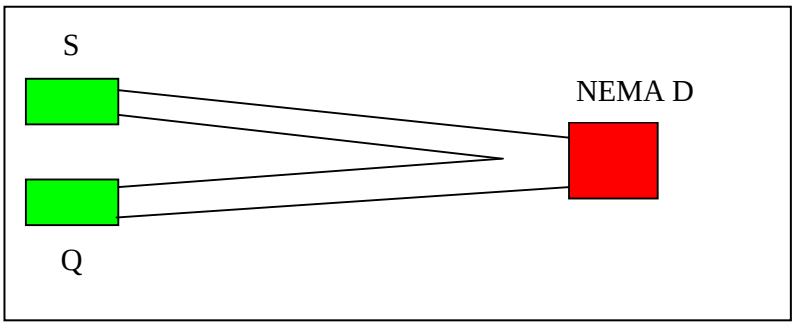

Figure 2 –Conversion Cable Schematic

The conversion cable should be wired as desired between the NEMA D connector and the S and Q connectors. Tables 1 and 2 are used to determine what CID data bits will be associated to what NEMA D inputs or outputs. It is important to map NEMA outputs to CID input bits, and NEMA inputs to CID output bits.

Table 1 - Pin Assignments for CID Output to NEMA D Connector

CID Output Bit	NEMA connector	NEMA pin	CPC Connector	CPC Pin
26	B	B	S	1
28	B	W	S	2
29	B	X	S	3
30	B	v	S	4
63	A	g	S	5
64	A	y	S	6

Table 2 - Pin Assignments for CID Intput to NEMA D Connector

CID Input Bit	NEMA connector	NEMA pin	CPC Connector	CPC pin
59	A	DD	Q	1
60	B	A	Q	2
61	A	e	Q	3
62	B	C	Q	4
63	B	t	Q	5
64	B	f	Q	6

Table 3 - S and Q connector part numbers

Qnty.	Connector	Description	AMP P/N	Digi-Key P/N
1	S	CPC, 13-9, Standard, Plug	206708-1	A1302-ND
1	Q	CPC, 13-9, Standard, Free Hanging Receptacle	206705-2	A1352-ND
2	S and Q	Cable Clamp	206966-1	A1331-ND
10	Q	Series 1 Contacts, Pin	66103-4	A1342-ND
10	S	Series 1 Contacts, Socket	66105-4	A1343-ND
100'	n/a	24 ga. stranded copper hookup wire	n/a	n/a

Part 2: Traffic Real-time Hardware-in-the-loop Simulation Performance Analysis

Abstract—The controller interface device (CID) was designed to support traffic hardware-in-the-loop real-time simulation. To reduce the probability that expected or unexpected combinations of data values or event sequences could cause real-time systems to miss deadlines or add bias value to final results, performance analysis of a real time HILS has been conducted. The performance analysis focuses on the latency measurement, latency calculation, task number calculation, and HILS system bottleneck analysis. Three theories, Petri Nets, Rate Monotonic, and Queuing, were used to analyze the HILS system performance. Additionally, experiments were carried out to measure latency that occurs at various locations in the HILS system. This paper documents the HILS process, analyzes the latency errors in the system, and describes the conditions under which HILS with the CID can provide true real-time simulation of traffic control systems.

Introduction

Transportation engineers often use computer models to evaluate or test how well an arterial traffic system will operate under a given traffic demand. Depending on the complexity of the system under study, the engineer may use either a macroscopic deterministic model to evaluate the performance of a fixed-time signalized intersection or a stochastic microscopic simulation model to test the operation of actuated signalized intersections during over-saturated conditions. But one of the most serious limitations of the simulation models in common use today is that their signal controller emulator sub-models do not include many of the features present in today's traffic controller. The continuing changes to control algorithms as well as the need for manufacturers to maintain propriety precludes simulation software model developers from including these features into their models. For example, the FHWA TSIS/CORSIM model [1-2] uses actuated traffic control technology from the 1980's and does not allow the user to specify the advanced control algorithms that have been developed in recent years. Hardware-in-the-loop simulation (HILS) [3-6], and the controller interface device that makes HILS possible, were introduced into the traffic industry during the last 7 years [3]. While simulation programs such as CORSIM allow engineers to test signal timing plans; their controller logic does not include all of the advanced features available on today's controllers. With HILS, engineers can test a signal-timing plan using an actual traffic controller, in real time, in the convenience and security of the office or laboratory.

Background

This section will first introduce the concepts of the HILS and present the CID HILS system.

Hardware-in-the-loop Simulation (HILS)

The basic idea of the HILS technique is very simple. In order to test a piece of hardware, a special simulation model is used to generate test information, which is used to "fool" the hardware into thinking it is operating in a real environment. By doing this, all information collected from the hardware will tell the user how HILS works.

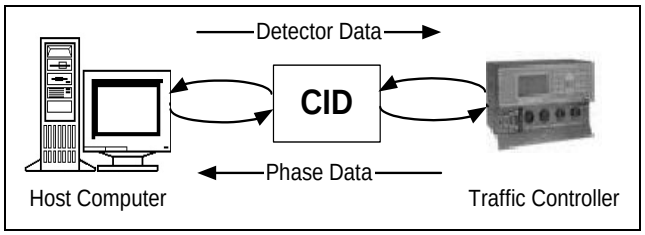

Figure 1: CID system architecture.

The CID (shown in the middle of Fig. 1) provides a link between a host computer and an actual NEMA, type 170 or 2070 [7-9] traffic controller. This link is known as real-time, hardware-in-the-loop. A real traffic signal controller replaces, when this link is used with a traffic simulation model, the traffic controller component or the simulation program's internal controller emulation logic. The CID allows a computer to communicate with traffic control hardware by allowing the simulation models to send detector actuations to the control device and to read phase indications back from the control device. The CID functions as a bridge between the electrical signals of the computer and those of the traffic signal controller.

Real-Time Systems

Real-time computer systems must produce correct outputs in response to real world inputs within well-defined deadlines [10]. Real-time systems can be divided into hard systems and soft systems. Hard real-time systems cannot tolerate any failures to meet deadlines. Soft real-time systems can tolerate some deadline failures and still function correctly. An example of the former is any real-time aircraft or missile flight control system. A deadline failure in either of these systems could cause loss of an aircraft or a missile. The CID HILS is an example of a soft real-time system. Here, an occasional missed deadline may affect some measures of effectiveness (MOE), but should not cause the system to fail or crash.

Real-time systems can be characterized as using either static scheduling or dynamic scheduling. A static scheduling system has been pre-calculated before it starts. A dynamically scheduled system schedules all of its tasks at unpredictable times and it may be possible to predict that deadlines will be met for a given subset of

tasks. In a statically scheduled system, it is possible to produce software systems with very convincing guarantees of deadline performance.

CORSIM uses one-second time step. All objects in the traffic system simulator are updated during each time step and the updates are guaranteed, so CORSIM will not miss any computation deadlines. The CID/CORSIM HILS system is a statically-scheduled system. If the HILS system is overloaded during some specific time step, it cannot finish its tasks on time. The subsequent scheduled time step will be disrupted and cannot be initiated at the pre-scheduled time. The latency caused by the overload is not acceptable. The latency and any other faults occurring during the simulation may be accumulated into CORSIM MOEs.

HILS Latency Aanalysis

Latency is often used to refer to any delay in a system that increases the real or perceived response time beyond the desired response time. In a HILS system, latency is a measure of how much time it takes for a packet of data to get from the CORSIM model to the traffic controller or from the traffic controller to CORSIM. While the expectation is that a data packet should be transmitted instantly between one point and another (that is, with no delay at all), this will not happen with real systems. The following sections introduce the sources of latency within HILS.

Introduction

Consider two simulation runs, one using CORSIM and the other using standard HILS (using CORSIM). Even if the traffic network geometry, random seeks and signal timing are the same for both simulation runs, the MOEs produced by the simulations may be different. Why should this be true? The reasons behind these differences are complicated and not always evident. From the traffic network perspective, these two simulations are the same. But from the systems point of view, the simulations are different. The main difference results from the signal control algorithm and detector response algorithm of the traffic controller (used as part of HILS) and CORSIM, the inconsistency of the running states of the traffic controller and CORSIM, and other random factors that occur during run time.

In fact, CORSIM and standard HILS represent two different simulation systems. CORSIM is a software-only simulation, with the signal timing control and detector call response algorithm included as part of the simulation software. Standard HILS is a real-time system with four components: the CORSIM model itself, the CID interface software, the CID hardware, and the traffic controller. The signal control and detector call response algorithms are accomplished through the interaction of these four components. When CORSIM runs as a stand-alone model, the phase updating and detector responses have no latency as they are all handled within the simulation model. However, the standard HILS implementation of these two algorithms requires the integration

of these four different components. First, the simulation model must slow down its execution time to real-time. Next, the phase and detector information needs to travel back and forth between the four components (shown in Fig. 2) of the standard HILS system. Latency occurs during the generation of the data and the communication of the data between the various HILS components.

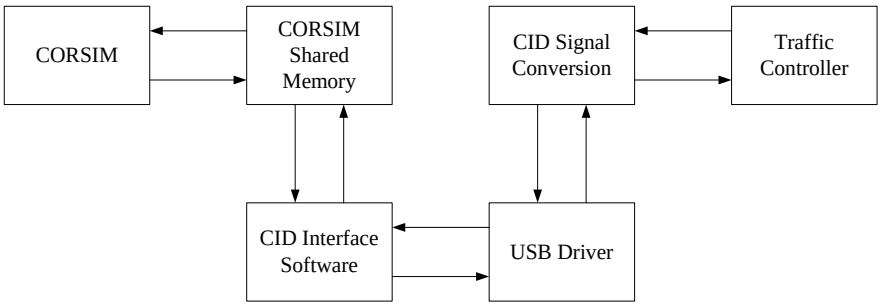

Figure 2: HILS data flow

Why Latency Exits

In real-world systems, such as a traffic controller, events happen continuously. In simulation models, either standard simulation models or HILS, events happen discretely. During standard HILS, CORSIM must complete all tasks for a given time step before the time step is over. For example, the CID interface module updates all phase and detector states within the time step. For any remaining time, the simulation engine sits idle. This could mean that the simulation model completes all required computations and data exchanges in less than a time step since the real-time simulation is not in reality a "real-time" simulation. If we want the simulation model to generate a better result, we need to decrease time step length in order to have a finer-scale resolution of system operations. And, simulation idle time should be avoided or be as small as possible. During simulation idle time, even if the model is doing nothing, the traffic controller will continually update its phase states. This can make the states of the simulated objects inconsistent. This inconsistency between the states of the simulated objects may introduce latency into the simulation results and differences in the MOEs that are generated between CORSIM and standard HILS.

The difference in the software algorithms is another reason for differences in the MOEs. The signal-timing algorithm used in CORSIM is different than the algorithms used in real traffic controllers that are used as part of HILS. The CORSIM signal control software module implements its internal signal timing and detector call response algorithms, while in HILS, the system signal timing and detector response algorithms are done by the interaction of the four components of HILS, including the traffic controller.

Because HILS is a real time system, there may be random factors introduced which cause latency and should be considered. Although HILS has error checking software modules, these random errors could be introduced into the MOEs produced by HILS.

HILS Latency Sources

The following section describes the potential for latencies in HILS and how these latencies might produce differences in the results generated by CORSIM and HILS. The factors that contribute to latency in HILS may include:

1) Propagation delay is the time that it takes for a data packet to travel between one place and another at the speed of light (in HILS, via the CID cable).
2) Transmission delay is the delay introduced by the medium itself. The size of the data packet affects transmission delay.
3) CID signal processing delay is the time that it takes for each CID to convert data from digital to analog or from analog to digital.
4) Software processing latency is the time that software modules take to complete their functions. In the CID system, the places in which software processing latency could be introduced are shown in the Fig. 2. The CORSIM shared-memory, the CID interface software, the USB driver, and the CID signal conversion and traffic controller firmware will introduce latency to the delivery of the CID data package.

The CID system includes two directions of data flow, as shown in Figure 2. The first is the detector data flow, which originates from CORSIM and travels to the traffic controller. The other is the phase data flow, which starts from the traffic controller and ends at CORSIM. Latencies can exist in both data flows.

Table 1 lists the factors that can contribute to latency in the HILS system. These factors can be categorized as either software latency or hardware latency. Software latency can result from four factors; while hardware latency can result from five factors (refer to Table 1).

Table 1: HILS Latencies Categories

	Latency Sources	
	Software Latency	Hardware Latency
CORSIM	√	
CORSIM Shared Memory	√	
CID Interface Software	√	
USB Driver	√	
USB Communication		√
CID Signal Conversion		√
Traffic Controller		√
Transmission		√
Propagation		√

The potential for software latency can be shown in an example of software-in-the-loop real-time simulation (SILS) (Figure 3) using the traffic controller emulator algorithm that resides in the interface software (only fixed time signal timing algorithm). It emulates the controller by calculating phase timing and sending the calculation result (phase information) back to CORSIM. In this case, the latency that may result can be divided into two parts. The first part occurs in CORSIM and is called CORSIM computation latency. The second part happens as part of the interface software and controller emulator and is called interface software computation latency. The purpose of conducting SILS controller emulator tests is to isolate the software and hardware latencies that could happen during SILS runs.

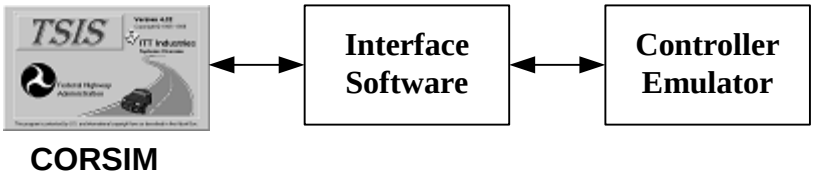

Figure 3: Software-in-the-loop simulation (SILS).

Experiments

In order to predict HILS performance, three experiments were first conducted to measure the three basic latencies that would be a part of an analysis of HILS performance.

USB 1.1 Communications Experiments

The latency due to USB communications is the time that data spends traveling through USB connections. In order to measure this latency, a USB Bus and Protocol Analyzer were used. The USB Bus and Protocol Analyzer is a device that captures, displays, and analyzes signals transmitted at various speeds through a USB port. This device is attached to the HILS host computer and is used as part of a special test developed to measure the latency of USB communication. A special test program was designed that can continuously read or write data to the USB bus. Three tests have been conducted with this program.

Test one was designed to test the latency of reading (IN) data from the USB. This test uses a special test program that keeps reading data from the CID continuously. The "ReadData" function can read data from a specific CID port or device. The program calls this function continuously.

Test two was designed to test the latency of writing (OUT) data to the USB. This test used a special test program that continuously writes data to a specific CID. This test program calls the "WriteData" function continuously.

Test three was designed to test the latency of the pair of actions of reading and writing. This test used a special program that continuously calls the "read" and "write" functions to the CID. When the first pair of function calls is completed, the second pair is called. This process continues.

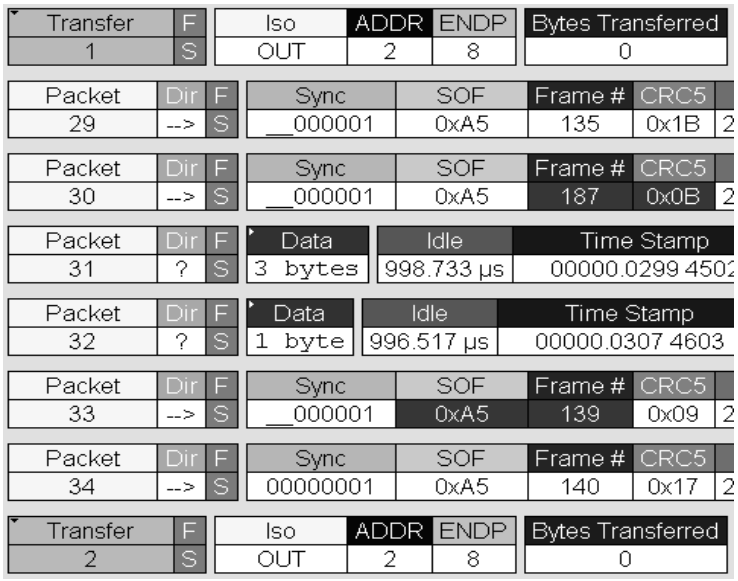
Transfer F Iso ADDR ENDP Bytes Transferred
1 S OUT 2 8 0
Packet Dir F Sync SOF Frame # CRC5
29 --> S __000001 0xA5 135 0x1B
Packet Dir F Sync SOF Frame # CRC5
30 --> S __000001 0xA5 187 0x0B
Packet Dir F Data Idle Time Stamp
31 ? S 3 bytes 998.733 µs 00000.0299 4502
Packet Dir F Data Idle Time Stamp
32 ? S 1 byte 996.517 µs 00000.0307 4603
Packet Dir F Sync SOF Frame # CRC5
33 --> S __000001 0xA5 139 0x09
Packet Dir F Sync SOF Frame # CRC5
34 --> S 00000001 0xA5 140 0x17
Transfer F Iso ADDR ENDP Bytes Transferred
2 S OUT 2 8 0

Figure 4: CID data package shown on the USB bus

The conclusion from these USB bus analysis experiments is that no matter what combination the function calls, there are six packets (Figure 4) between adjacent data transfers and the timing spent on these two successive function calls is a constant value of 6 ms. The USB bus utilization is low with all combinations of function calls. The USB data communication capability is sufficient for the HILS applications. The 6 ms value will not change with different host computer hardware platform and operating systems. This number is fixed for the

current Microsoft Windows operating systems. The USB communication latency will be accumulated linearly with the number of the external CIDs in the HILS system.

CID Hardware Signal Conversion Time Experiments

The CID firmware is the intelligence of the CID, and manages all hardware operations. The CID firmware employs a program interrupt method. When the interrupt handler grants an interrupt request, the CPU discontinues the normal flow of instructions and branches to a routine that services the source that requested the interrupt. When the subroutine finishes, execution resumes at the point where the interrupt occurred. The CID firmware has two interrupts, an internal and an external interrupt. The internal interrupts are the function interrupts that are used by the USB enumeration and the start of fame (SOF) interrupts for the operational data transfer [16]. The external interrupt is used by the self-test operation, which is initiated by the CID self-test button.

The USB is a polled bus. The USB host initiates all data transfers and keeps the bus constantly active by sending a SOF packet every 1 ms in isochronous mode. The available bus bandwidth is shared between simultaneously connected devices within the 1 ms time interval. The USB has a new frame every 1 ms, and therefore the SOF interrupt is also generated every 1 ms. One of the most important design issues for the CID is to make sure that all instructions in the SOF interrupt service routine are accomplished and the CID returns to the idle loop within 1 ms.

The CID firmware maintains an endless idle loop waiting for the operational data transfer interrupt to occur. When the interrupt occurs, the idle loop operation is stopped and the operational data transfer interrupt subroutine is executed. After the operational data transfer interrupt subroutine finishes, the CID firmware will return to the idle loop. The job of the operational data transfer interrupt subroutine is to update the “receive” and “transmit” FIFO queue (the CID firmware data buffer). This updating process is repeated every 1 ms. This means that the CID input and output update rate is 1 ms. The CID hardware signal conversion latency is therefore 1 ms.

Software Computation Time Experiments

CORSIM Computation CPU Time

CORSIM generates a log file after it completes a simulation run. This log file reports the total CPU time for the run. Table 2 tabulates the CORSIM calculation CPU time for five 120-minute test simulation runs (Table 2 1 to 5 column). The following equation is used to compute the average CORSIM calculation time for one simulation time step.

$$t = \frac{C}{T}$$

Where

t: average CORSIM computation time for one time step,
C: total CORSIM computation time,
T: CORSIM simulation time step (1000 ms).

For these experiments, the total CORSIM computation CPU time of 32 seconds (for 7200 seconds simulation period) is the maximum value. The values from experiments can be different for the different test runs even under the same test environment. The result of the calculation (320000/7200 ≈ 5 ms) is an estimate of the CORSIM calculation time for one simulation time step. The CORSIM calculation CPU time depends on the complexity of test simulation network, computer platform, and CORSIM versions.

Table 2: CORSIM Calculation CPU Times (Seconds)

Control Mode	Vol.	CORSIM Computation Time (C)				
		1	2	3	4	5
Fixed Time	High	22.28	20.84	17.48	18.18	19.45
	Low	18.29	16.76	14.62	15.68	15.00
Actuated	High	31.19	26.15	22.26	24.67	25.6
	Low	24.7	23.9	20.22	21.1	20.89
Coord. .Fixed Time	High	25.67	20.18	16.86	18.78	18.45
	Low	21.03	16.61	14.14	15.52	15.57
Coord. Actuated	High	31.17	26.19	23.21	25.02	25.95
	Low	27.72	23.49	20.06	21.9	20.73

Interface software computation CPU time

In the HILS tests, CORSIM also generates a log file after it completes a simulation. This log file reports the total CPU time for the run. Table 3 tabulates the HILS calculation CPU time for five 120-minute test runs. The following equation is used to compute the CORSIM calculation time for one time step.

$$t = \frac{H - C - T}{T}$$

Where

t: interfaces software computation time,
H: total HILS computation time,
T: CORSIM simulation time step (1000 ms),
C: total CORSIM computation time (for this case, 32 seconds).

For example, the total HILS computation time (7380 seconds) is the maximum value from experiments. This HILS computation time includes the interface computation time, CORSIM computation time, and the 7200 seconds delay time. The values from experiments can be different for different test runs even under the same test environment. The result of the calculation (7380-7200-32/7200 ≈ 21 ms) is a latency estimated value for one simulation time step. This experiment value has two components. One is interface software latency. The

other is USB communication time for one detector data and one phase data package. So the average interface software latency is 21-2×6 = 9 ms.

Table 3:HILS calculation CPU times (seconds)

Control Mode	Vol.	HILS Computation Time (H)				
		1	2	3	4	5
Fixed Time	High	7379.6	7379.7	7379.1	7378.6	7378.9
	Low	7378.6	7378.9	7379.4	7378.8	7378.8
Actuated	High	7379.4	7379.1	7379.1	7379.6	7379.7
	Low	7378.8	7380.2	7379.0	7378.6	7379.2
Coord. .Fixed Time	High	7379.1	7378.5	7378.7	7378.7	7379.2
	Low	7379.1	7379.0	7378.5	7379.0	7379.3
Coord. Actuated	High	7378.7	7379.2	7378.7	7378.8	7378.7
	Low	7378.7	7379.3	7379.3	7379.1	7379.2

Performance Analysis

A theoretical model uses mathematical constructs to describe the behavior of a real-time system. These analytical approaches often make simplifying assumptions about certain aspects of the system. Common analytical methods for verification of real-time systems include schedule-ability analysis, systems modeling, and stochastic and queuing theoretical analysis. Such theoretical analysis methods can provide quantitative information about certain properties of a system.

Rate Monotonic scheduling [12], Petri Nets [13], and Queuing theory [14] are three of the most widely used approaches in the analysis of real-time systems performance. These three methods were used to analyze the performance of the CID real-time system.

Predictability is an important issue for the analysis of the performance of real-time systems. The CID real-time system is a static scheduling system. The rate monotonic scheduling algorithm can be used to calculate the maximum task number for a CID real-time system. The CID real-time system maximum task number is the number of the external-control intersections. This number is the current limitation on the number of CIDs that can be used simultaneously in the HILS system.

Petri Nets is a graphical and mathematical modeling tool. It can be used to describe and study systems that are characterized as being concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic. It can be used to simulate the dynamic and concurrent activities of systems. Petri Nets can be built to model the behavior of real-time systems.

Queuing models are often used to assess the performance of real-time systems for which average or expected throughput under specified load conditions is desired. These models may be useful in order to understand CID real-time system operation if deadlines are guaranteed by other logic. The performance predictions of queuing models for CID systems may serve both as initial feasibility confirmation and for later testing.

Introduction to Rate Monotonic Scheduling Analysis

For a set of tasks, if their resource and priority conflicts are resolved, each task is completed prior to the time that it is scheduled for another execution, and CPU utilization (ratio of active CPU time to active plus idle time) is less than 1.0. These tasks are schedulable. If, in a system, all its tasks are schedulable, and all its tasks will be completed successfully before their deadlines if executed on schedule, this system meets its deadlines.

For a dynamically scheduled system with unpredictable task phasing (task start times in some relation to each other) and cycle times (period of task repetition), schedulability is hard to prove using a theory. A statically scheduled system, which does not stop, executes its tasks cyclically after an initial startup period. For such systems it can be shown that an independent task set (Ω:τi$\in\Omega$) is schedulable by the rate-monotonic scheduling algorithm [12], if

$$\frac{C_1}{T_1}+\frac{C_2}{T_2}+...+\frac{C_n}{T_n}\le U(n)=n(2^{\frac{1}{n}}-1)\le 1.$$

Where

n : Number of tasks.

C_i : Execution time for task i, second.

T_i : Period of time for task i, second.

$U(n)$: Least-upper-bound CPU utilization for n tasks.

The assumptions behind this expression are that task execution times and period times are known, and tasks are not required to have the same period. The rate monotonic schedulability relation above is optimal in the sense that no other fixed-priority scheduling algorithm can schedule a task set that the rate monotonic algorithm fails to schedule [15].

The CID system has four hardware resource components: the computer, the USB, the CID, and the traffic controller. Among these four components, only the computer and the USB are the resources for which utilization needs to be considered. Their utilization rate cannot be higher than one.

CORSIM and the CID interface software require CPU resources. While the CID real time system is running, CORSIM will only be initialized for one instance. This means that only one instance of CORSIM will be initialized. Thus there is no scheduling problem for the CORSIM software.

USB Resource Scheduling Calculation

The USB is a resource for which utilization needs to be considered. For each one-second CORSIM time step, the CID interface software has to read in one package of phase data from the USB and send out one package of detector data to the USB for each external controlled node (detector data package only for the actuated node).

If there are n externally controlled nodes for a traffic network, there will be 2n data packages requiring USB resources. Under ideal conditions, the execution time for the data package is 6 ms. The period of time for each data package is $1000 - C_t - I_t + 2t$ (ms). The experimental value I_t has two components. One is interface software latency. The other is USB communication time for detector and phase data. The CORSIM computation C_t is an estimation value when there are 80 nodes in the traffic network. The maximum number of tasks is calculated as:

$$\frac{(t)_1}{T_1} + \frac{(t)_2}{T_2} + ... + \frac{(t)_{2n}}{T_{2n}} \le 1.$$

$$n \le \frac{T}{12}.$$

$$T = T_1 = T_2 = ... = T_{2n} = 1000 - C_t - I_t + 2(t).$$

$$n \le \frac{1012 - C_t - I_t}{12}.$$

Where

C_t: CORSIM calculation time, ms,

I_t: CID interface software calculation time, ms,

n: Maximum number of tasks,

t : Data package USB communication latency (6 ms).

From the above calculation, the maximum number of CIDs that can be included in the CID real-time system depends on the CORSIM and interface software latency. If CORSIM and the interface software latency is zero (which is impossible), then n =1000/12 ≈ 83. This number is the absolute limit of the number of CIDs that can be connected to the real-time system.

For our test cases, $C_t = 30$ ms (from Table 5) and $I_t = 21$ ms, then n ≈ 80. This number is not fixed. It depends on the experimental hardware (computer) environment and complexity of the traffic network under consideration.

CPU Resource Scheduling Calculation

The CPU is a resource for which the utilization problem must be considered. CORSIM and the interface software must finish their execution within the one second time step. Suppose there are n externally controlled nodes for a traffic network. The detector and phase data update for one externally controlled node is one task. Thus there will be n tasks requiring CPU resources. From this we have:

$$\frac{(2t)_1}{T_1} + \frac{(2t)_2}{T_2} + ... + \frac{(2t)_n}{T_n} \le 1.$$

$$T = T_1 = T_2 = ... = T_n = 1000.$$

$n \le \frac{T}{2t}$.

For our test cases, then n=1000/12=83.

The USB and computer CPU resource scheduling calculation identifies that the CID HILS bottleneck is USB. The maximum number of the CIDs that could be used in the HILS is 80.

Modeling HILS System Using Petri Nets

A Petri Net [13] is a modeling tool that can be used when building real-time systems. It can serve as a performance predictor during initial model development stages, and later as a guide for validating the system. It is well suited for the study of discrete event systems. It can model and analyze systems with parallel and concurrent processes and validate the design and prototype of a new system.

The HILS system consists of the detector system and the phase system. The token for the detector system is the detector data package. The token for the phase system is the phase data package. The detector data package originates from CORSIM and ends at the traffic controller. The phase data package originates from the traffic controller and ends at CORSIM. The traffic controller and CORSIM connect these two systems into one closed-loop system (shown in Figure 5). There are two types of components in this closed-loop. The first type is the multi-copy component: the CID and the traffic controller. Each set of multi-copy components has one CID and one traffic controller. In the HILS system, multi-copy components are running independently and in parallel. The other type is the single-copy component: the host computer, CORSIM, the interface software, and the USB. The single-copy component runs in sequence and dependently.

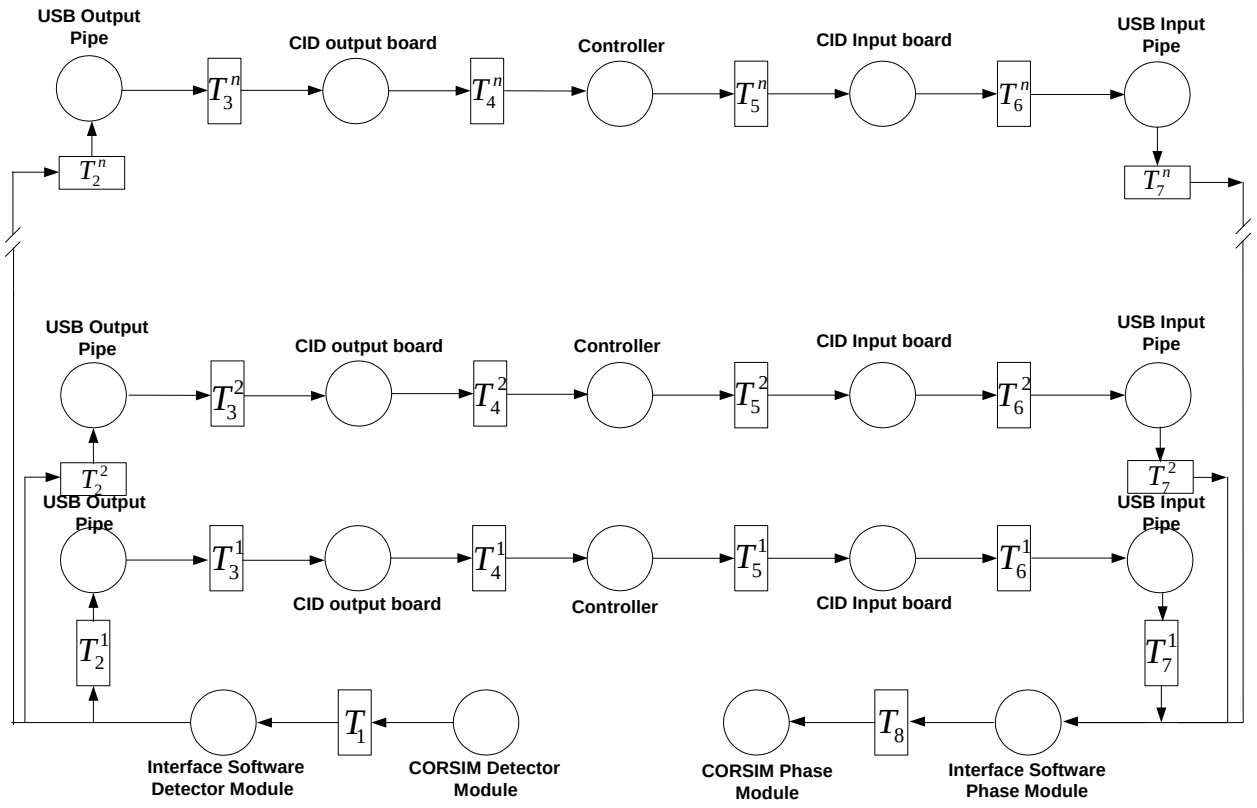

Figure 5: HILS system Petri Net

For the HILS environment, each time step should be guaranteed that it would not be overloaded. The overload means that within every 1000-millisecond time step, total delay is greater than time step.

$$Z = \sum_{i=1}^{8} T_i \le 1000.$$

$$Z = \sum_{i=1}^{8} \overline{T_i} + \sum_{i=1}^{8} \Delta t_i .$$

$$E(Z) = \sum_{i=1}^{8} \overline{T_i} .$$

$$Var(Z) = E(\sum_{i=1}^{8} \Delta t_i)^2 = \sum_{i=1}^{8} E(\Delta t_i)^2 + 2E\left[\sum_{i \ne j}^{i,j=1...8} \Delta t_i \Delta t_j \right].$$

Where

Z : Total latency for every time step,

T_1 : Total CORSIM detector data computation time,

T_2 : Total interface software detector data computation time,

T_3 : Detector data package USB communication time,

T_4 : CID detector digital signal to analog signal conversion time (1 ms),

T_5 : CID phase analog signal to digital signal conversion time (1 ms),

T_6 : Phase data package USB communication time,

T_7 : Interface software phase data computation time,

T_8: CORSIM phase data computation time,

$\overline{T_i}$: mean value for I component,

Δt_i: The error for T_i's,

$E(Z)$: The mean value of total latency,

$Var(Z)$: The variance of total latency.

$T_1, T_2, T_3, T_4, T_5, T_6, T_7,$ and T_8 are statistically independent, $E(\Delta t_i \Delta t_j) = 0,$ where $i \neq j$.

Then

$$Var(Z) = \sum_{i=1}^{8} E(\Delta t_i)^2.$$

In order to compute Z and Var(Z), a set of experiments was conducted. The timing components in Table 4 were measured as part of the experiment.

Table 4: Timing Components in the HILS Process.

	CORSIM computation time (T_1 and T_8) (ms)	Interface Computation time (T_2 and T_7) (ms)	Detector data communication time (T_3) (ms)	Phase data communication time (T_4) (ms)	Phase signal conversion time (T_5) (ms)	Detector signal conversion time (T_6) (ms)
Mean	3.854875	0.053811	5.933845	5.730484	1	1
Stdev	0.261182	0.001018	0.039391	0.032673	0	0
Variance	0.068216	0.000001	0.001552	0.001068	0	0

In Table 4, the sum of all timing components for each CID number in the HILS is computed. The second column "k" is a factor that measures the CORSIM and interface software computation time increment with the number of CIDs in HILS. Two experiments were conducted to measure this factor (with the number of CIDs equal to 1 and 40). Suppose:

$$T^n_{(1+8)} = k \times n \times T^1_{(1+8)}.$$

$$T^n_{(2+7)} = k \times n \times T^1_{(2+7)}.$$

Where

n: The number of CIDs.

k: The slope.

$T^n_{(1+8)}$: The sum of T_1 and T_8 (CORSIM computation time), when the number of CIDs is n.

$T^n_{(2+7)}$: The sum of T_2 and T_7 (Interface computation time), when the number of CIDs is n.

When the number of CIDs in HILS equals 1, $T^1_{(1+8)}$ and $T^1_{(2+7)}$ were recorded and "k" was measured. When the number of CIDs in HILS equals 40, $T^{40}_{(1+8)}$ and $T^{40}_{(2+7)}$ were recorded and "k" was measured. Suppose the relation between "k" and the number of CIDs is linear. The slope of the linear relation is easy to calculate. The value of

"k" is shown in Table 5. Because the CID runs in parallel, the detector data and phase data conversion time are not increased while the number of CIDs in HILS increases.

Table 5: Sum of all components timing (ms).

Number Of CIDs	k	CORSIM Computation Time	Interface Computation Time	Phase Data Communication Time	Detector Data Communication Time	Detector Data Conversion Time	Phase Data Conversion Phase	Sum
1	**1**	**3.8549**	**0.05381**	**5.9338**	**5.7305**	**1**	**1**	**17.5730**
10	1.8333	7.0673	0.09865	59.3385	57.3048	1	1	125.8092
20	2.6666	10.2797	0.1435	118.6769	114.6097	1	1	245.7097
30	3.5	13.4921	0.1884	178.0154	171.9145	1	1	365.6103
40	**4.3333**	**16.7045**	0.2332	237.3538	229.2194	1	1	485.5108
50	5.1666	19.9169	0.2780	296.6923	286.5242	1	1	605.4113
60	6	23.1293	0.3229	356.0307	343.8290	1	1	725.3119
70	6.8333	26.3416	0.3677	415.3692	401.1339	1	1	845.2124
75	7.25	27.9478	0.3901	445.0384	429.7863	1	1	905.1626
80	7.6666	29.5540	0.4126	474.7076	458.4387	1	1	965.1129
81	7.75	29.8753	0.4170	480.6414	464.1692	1	1	977.1030
82	7.8333	30.1965	0.4215	486.5753	469.8997	1	1	989.0930
83	7.9166	30.5178	0.4260	492.5091	475.6302	1	1	1001.083

Latency Analysis Using Queuing Theory

The CID real-time queuing system consists of one server and customers. The CID queuing system server is the USB, which handles the customers (phase and detector data packages). A customer is a data package to be processed by the USB. The CID real-time system has only one server. When multiple data packages are waiting for service by the USB, a queue forms. The CID data packages arrive for service, wait for service if it is not available, and leave the system after being served.

The CID real-time system has two queues, phase data and detector data. The inputs for these two queues are the phase and detector data packages, which are serviced by the USB in sequence. The CID queue service discipline is first-in-first-out (FIFO). When the CID USB queue server is busy, no data package will arrive. When the USB server is idle, the data package will be fetched from the data buffer and be serviced by the server. As soon as the data package leaves the USB, the next data package will arrive and be serviced. So the CID real-time system queues arrival rate is equal to the service rate. The queuing pattern for the CID real-time system is D/D/1 [14], which means the arrival rate and service rate are constant and the number of servers is 1. Figure 6 shows the CID queues. During each simulation time step, there are two queues. The interface software calculation time, the phase data queuing time, and the detector data queuing time make up the major parts of the interface software latency.

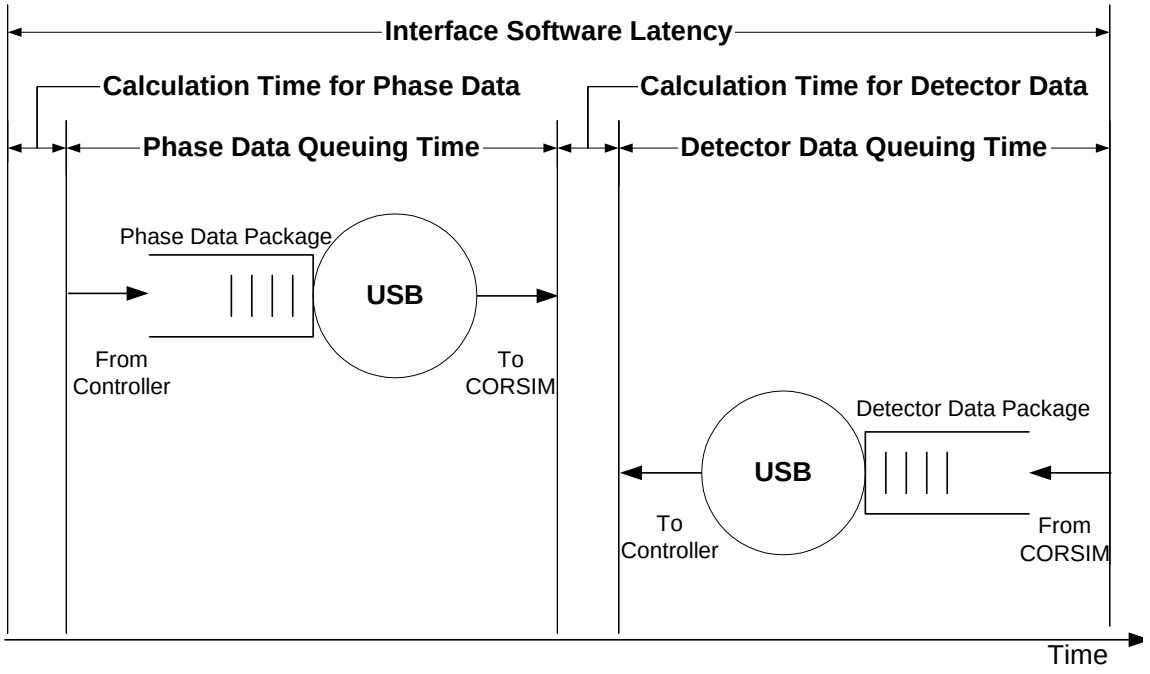

Figure 6: Queues in the CID system.

The CID queue has only one server and a limited number of waiting positions. Let the arrival process of phase data be such that arrivals come at rate l. Assume that the average service rate is μ (with the mean service time of 1/μ or 6 ms. The arrival rate l is equal to the service rate μ). The mean time spent in the system is:

$$W = \frac{1}{\mu}.$$

The total time for the CID data package to be served is:

$$T_i = i \times W = \frac{i}{\mu}.$$

Where i = 1, 2, …, 81.

Phase Data Package Queue

During each time step, the phase data package data queue arrives in the USB data buffer. The CID generates these arrival data packages independently and in parallel. The phase data from the traffic controller is an analog signal. The CID converts these analog signals to one digital data package. The CID signal conversion time is 1 ms, so it can be assumed that all phase data packages arrive at the USB for service within 1 ms. The request from interface software determines arrival of the phase data package. The phase data package mean time spent in the system is: W=1/μ=6 ms.

Detector Data Package Queue

During each time step, CORSIM will update the detector status for the externally-controlled nodes and saves the detector status information to the TSIS shared memory. After the CID interface software is executed, it reads all detector data from the TSIS shared memory and sends them to the USB. The detector data package mean time spent in the system is: W=1/μ=6 ms. The total service time (or latency) for 40-83 CIDs is calculated as shown in Table 6:

Table 6: CPU Resource Distribution for One Simulation Time Step

External control nodes number	Phase data package latency (ms)	Detector data package latency (ms)	Total latency (ms)	Sum of all components timing (ms)	Time left for CORSIM calculation (ms)
40	240	240	480	485.51	520
50	300	300	600	605.41	400
60	360	360	720	725.31	380
70	420	420	840	845.21	160
80	480	480	960	965.11	40
83	498	498	996	1001.0	4

Summary

The CORSIM model has a one-second simulation time step, meaning that within this one-second period, each software module must complete all of its tasks. The three software modules are: the CORSIM simulation engine, the CID interface software, and the USB driver. They all could introduce latency into HILS.

Latency in the HILS system can be categorized as either software latency or hardware latency. Software latency can result from CORSIM, CORSIM shared memory, CID interface software, and USB driver. Hardware latency can result from USB communication, CID signal conversion, traffic controller, and signal transmission.

The experimental results show that (1) there are 6 ms latency between two consecutive USB data transmission. (2) CID hardware need 1 ms to convert data from analog to digital or from digital to analog. (3) CORSIM simulation engine and interface software execution time is not a major part of the latency. Most of HILS latency comes from USB communication.

The CORSIM simulation engine execution time depends on the characteristics of the traffic network. The larger the traffic network, the more time the CORSIM simulation engine needs for its calculations. The CID interface software module and the USB driver execution time depends on the number of external nodes. The larger the number of the externally controlled nodes, the more execution time the interface software module and USB driver will need.

Rate Monotonic scheduling, Petri Nets, and Queuing theory were used to analyze the performance of the CID real-time system. These mathematical models give similar quantitative information about HILS system. For one second time step simulation models, the upper limit number of CIDs that can be used in the HILS is around 80. Table 6 shows the results of the calculations of the CPU resource distribution for several cases. When there are 83 externally controlled nodes in a traffic network, there is only 4 ms of execution time left for the CORSIM calculations. This time is not enough for the other tasks. When there are 40 externally-controlled nodes in a traffic network, there is 520 ms of execution time left for CORSIM calculations. The 520 ms CORSIM calculation time for each 1000 ms time step is sufficient for most cases.

From the above analysis, when the externally controlled nodes number or CID number equals 40, the computation time left for CORSIM is nearly half of one time step. Factors like the complexity of the traffic network, the number of vehicles, the number of intersections in the traffic network, and the signal control mode, will affect the CORSIM computation time. For different cases, even the same case running on different machines, the CORSIM computation time will be different. Adequate CPU resources should be reserved for the CORSIM computation.

Conclusion

This paper documented the HILS process (including all data transfers between the traffic simulation model, the CID, and the traffic controllers) and analyzed the errors (time latency) introduced into the system. After this

analysis, it is concluded that if the number of CIDs in the HILS system is less than 40 for one second simulation time step, HILS can provide true real-time simulation with minimal or no latencies that would adversely affect the simulation results.

The existing CID USB communication latency is 6 ms. This value will not change with host computer or operating system. From the performance analysis that focuses on the latency measurement, latency calculation, and task number calculation, USB is identified as HILS system bottleneck. If, in the future, the CORSIM time step upgrades to 100 ms, the existing HILS system can only test up to four CIDs at the same time. In order to overcome this limitation, a new CID with less communication latency is needed.

References

[1]. ITT Systems & Sciences Corporation. CORSIM User's Manual. Version 1.04. FHWA, U.S. Department of Transportation, March 1998.

[2]. ITT Industries, (2003). CORSIM Run-Time Extension (RTE) Developer's Guide, Version 5.1. Contract No. DTFH61-01-C-00005, February 2003.

[3]. Bullock, D., and A. Catarella. A Real-Time Simulation, "Environment for Evaluating Traffic Signal Systems." Paper presented at the 77th Annual Transportation Research Board Meeting, Washington D.C., January 1998.

[4]. Engelbrecht, R.J., K.N. Balke, S.P. Venglar, S.R. Sunkari., "Recent Applications of Hardware-in-the-Loop Traffic Simulation." Compendium of Papers of the 70th Annual Meeting of the Institute of Transportation Engineers (CD-ROM), Institute of Transportation Engineers, Washington D.C., 2000.

[5]. Bullock, D., B. Johnson, R. Wells, M. Kyte, and Z. Li, "Hardware-in-the-loop simulation," Transportation Research Part C: Emerging Technologies, Vol 12, Issue 1, pp. 73-89, February 2004.

[6]. Wells, R. B., Fisher J., Zhou Y., Johnson B. K., Kyte M., "Hardware and Software Considerations for Implementing Hardware-in-the-Loop Traffic Simulation". IECON'01: The 27th Annual Conference of the IEEE Industrial Electronics Society. Nov. 29-Dec. 2, 2001, pp. 1915-1919.

[7]. National Electrical Manufacturers Association. "Traffic Control Systems. NEMA Standards" Publication No. TS 1-1989. Washington, D.C., 1989.

[8]. National Electrical Manufacturers Association. "Traffic Controller Assemblies. NEMA Standards" Publication No. TS 2-1992. Washington, D.C., 1992.

[9]. California Department of Transportation, "Transportation Electrical Equipment Specifications (TEES)." Sacramento, CA, March 1997.

[10] Burns, A. and Wellings A., (1990). "Real-Time Systems and Their Programming Languages." International Computer Science Series. ISBN: 0201175290. Addison-Wesley Publishing Company. 01 January, 1990.

[11] Zhou, Y. and R. Wells, (2000). "A USB Compatible Controller Interface Device." Masters' degree thesis. University of Idaho. 2000.

[12] Sha, L., Rajkumar, R., and Sathaye, S. S., (1994). "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems." IEEE Proceedings, January 1994.

[13] Peterson, J. L. Petri Nets. ACM Computing Surveys, Vol. 9, No. 3, pp. 223–252, September 1977.

[14] May, D. Adolf. (1990) "Traffic Flow Fundamentals." University of California, Berkeley. ISBN: 0-13-926072-2. Copyright 1990 by Prentice-Hall, Inc. A Division of Simon & Schuster Englewood Cliffs, New Jersey 07632. P338.

[15] Liu, C.L. and Layland, James W., (1973), "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." Journal ACM, V. 20, No. 1, pp. 46–61. January, 1973.

[16] Zhou, Y. and Wells, R., (2000). A USB Compatible Controller Interface Device. Masters' degree thesis. University of Idaho. 2000.

Part 3: A Method of Verifying the Accuracy of Real-Time Hardware-in-the-Loop Traffic Simulation

Chapter 1 – Introduction: Real-Time Hardware-in-the-Loop Traffic Simulation

1.1 Traffic Control Systems

Traffic signals are controlled by traffic controllers, embedded computers that set light scheduling according to a programmed algorithm. Traffic controllers vary widely in intelligence; they may implement simple fixed-time scheduling systems, more complex traffic-actuated control, or advanced interconnected control systems.
A traffic controller's control outputs are called phase indications; they show allowed movement for vehicles or pedestrians in a certain path. Traffic-actuated controllers have inputs from traffic detection sensors (usually inductive loops) and/or from pedestrian call buttons. Phase decisions in an actuated controller are made based on metrics extracted from the input data, including the presence of waiting vehicles, vehicle speed, and traffic volume or density [1].

1.2 Traffic Simulation

Traffic engineers frequently use computer "micro-simulation" tools to design and tune traffic systems. (A micro-simulator is one that models the behavior of individual vehicles in the system.) Common simulators include CORSIM (CORridor SIMulation), developed by the Federal Highway Administration as part of its Traffic Software Integrated System package, and VISSIM (a German acronym; the name means roughly "traffic in towns simulation"), developed commercially by Innovative Transportation Concepts, Inc.; other commercial simulators are also available, but less widely used at present. These simulators typically provide measures of effectiveness (MOE) for the simulated system, such as total vehicle delay, stopped delay, and queue lengths [2]; detailed run results; and an animation of the system as it is being simulated. Simulations are based on stochastic vehicle models, but are repeatable for a given random seed.

1.3 Hardware-in-the-Loop Simulation

The utility of traffic simulation is reduced by the fact that the algorithms in modern traffic controllers (the devices that actually control traffic signals) are often complex and proprietary, and cannot usually be duplicated in the simulation model. The idea of hardware-in-the-loop (HIL) traffic simulation was developed to alleviate this problem. In a HIL simulation, the simulator is run in real-time (one simulated second == one actual second), and a hardware interface provides real-time interaction between the simulator and one or more traffic

controllers. On each time step, phase data are recorded and transmitted to the simulation, and traffic detector pulses generated by the simulation program are transmitted to the appropriate traffic controllers.

1.4 The Controller Interface Device

Several devices have been developed to manage the necessary interface in a HIL simulation between the computer and one or more traffic controllers, including models designed at Texas A&M University [3], Louisiana State University [4], and the National Institute for Advanced Transportation Technology (NIATT) at the University of Idaho [1]. This thesis will focus on NIATT's "CID II" model, which is presently entering commercial production by McCain Traffic Supply, Inc.

The CID II connects to a personal computer using the Universal Serial Bus (USB) 1.0 interface. It provides discrete digital inputs and outputs (64 of each) for connection to a traffic controller by means of a large, unsightly cable. A microcontroller handles timing of the CID's outputs (corresponding to traffic detector pulses), removing some of the burden of managing timing from the computer's decidedly non-real time operating system.

The USB specification allows up to 127 devices to be connected to a single computer. Practically, the number of CID II devices that can be used in a simulation is limited by communication latency.

1.5 Communication Latency and Simulation Accuracy

It was originally expected, based on bandwidth calculations, that a computer could communicate with around 40 CIDs in one millisecond. This has proved not to be the case, due to software limitations; in fact, it requires 10 ms to send data to and receive data from *one* CID. In a simulation with only a few CIDs, this should be insignificant, since the simulation time step is usually 1000 ms long. However, in simulations with tens of CIDs, this delay could approach the size of the time step.

In general, a onetime step timing error does not seem significant; in most simulation systems, time step frequency is chosen well above the maximum transient frequencies. However, there is doubt as to whether a 1000 ms time step length is small enough for advanced traffic control systems [5], and commercial simulators are moving towards smaller time steps (typically 100 ms).

The most relevant study of CID timing issues was undertaken at Louisiana State University with a different type of CID [4]. Results (MOEs) from a number of hardware-in-the-loop simulations for both fixed-time and traffic-actuated controllers were compared to results from "normal" (software-only) simulations and found to have no statistically significant deviation. However, this *type* of study is not as satisfactory in general as might be hoped: it can only compare results for traffic controllers that can be adequately modeled in software; in fact, there is little need to use HIL simulation with such controllers. Because there is by definition no easy way to model the

operation of traffic controllers with proprietary or highly complex algorithms, this evaluation method cannot determine the impact of the CID interface on them.

1.6 Verifying the Accuracy of Hardware-in-the-Loop Simulation

This thesis will present the development of a software/firmware tool to aid in evaluating the impact of communication latency on hardware-in-the-loop simulation with the CID II for a variety of traffic systems and controllers, including those that cannot be modeled in software. In addition, as CID-related research continues at the University of Idaho and latency issues change, it will provide a framework for evaluating the impact of timing changes.

Chapter 2 – Controller Interface Device Architecture

The controller interface device (CID) developed by the National Institute for Advanced Transportation Technology, "CID II," is a key component in the hardware-in-the-loop traffic simulation research being conducted at the University of Idaho, and in this thesis' topic in particular. This chapter will provide an overview of CID II's architecture. (Detailed information can be found in Ying Zhou's M.S.E.E. thesis [1]; although written about the earlier "pre-production prototype" CID, most information is still relevant.)

2.1 Overview

CID II provides 64 digital inputs and 64 digital outputs, which are connected to a traffic controller with a large custom-built cable. Inputs and outputs are electrically compatible with several of the most popular traffic controllers. Output pulse lengths can be determined with 1 ms resolution (with some restrictions). The computer interface is provided via the Universal Serial Bus (USB) 1.0 protocol, theoretically allowing up to 127 CIDs to be connected to one computer.

The CID uses a motherboard/daughterboard layout to facilitate easy assembly and part replacement. Seven daughterboards are used: microcontroller, display, input (2), output (2), and power supply [6].

2.2 Microcontroller

The microcontroller used is Cypress Semiconductors' EZ-USB, a 24 MHz 8051 variant with a USB interface core that automates many of the details of handling the interface. Program memory is off-chip in a 16 KB EPROM; data memory is on-chip, totaling over 6 KB of "xdata" (16-bit addressable) space. Communication with the input and output boards is accomplished over an 8-bit data bus with an enable line for each board.

2.3 USB Protocol

The Universal Serial Bus protocol allows up to 127 devices to be connected to a personal computer. Several different data transfer modes are provided to support different types of devices. The CID uses the isochronous transfer mode, which guarantees bounded transfer latency.

USB transfers occur in one millisecond long "frames." This provides a very convenient timing reference for the CID II. According to specification, it should be possible to communicate with around 40 CIDs in a single frame using the isochronous transfer method. In practice, it is difficult, if not impossible, to do so. The available USB driver can only write data to one device per function call, and each write call actually requires 6 ms to execute. The driver can *average* one transfer per frame, if it is passed a number of packets of data for a particular device, but this is not useful for the purpose of hardware-in-the-loop simulations, in which there is a relatively large time gaps between each packet.

This communication latency imposes a maximum practical limit of 100 CIDs in a simulation with 1000 ms time steps, or 10 CIDs if 100 ms time steps are used.

2.4 Traffic Controller Interface

The CID II uses discrete connectors and custom cables to interface with a variety of traffic controller models with up to 64 inputs and 64 outputs. Traffic controller inputs are read every 1 ms when the USB start-of-frame interrupt is received.

2.5 Firmware

The EZ-USB microcontroller's firmware is based on Cypress Semiconductor's "Frameworks" template, which, in conjunction with the EZ-USB's interface hardware, provides all basic USB functionality, as well as function hooks for USB interrupts. Most of the firmware's work is done in the start-of-frame (SOF) interrupt service routine (ISR), triggered by USB every 1 ms (and synthesized by the microcontroller when it is disconnected from USB).

From the firmware perspective, USB access simply consists of reading from or writing to "first-in-first-out" (FIFO) registers after each start-of-frame interrupt. Two data formats are defined: one for data sent to the CID, and one for data sent to the computer.

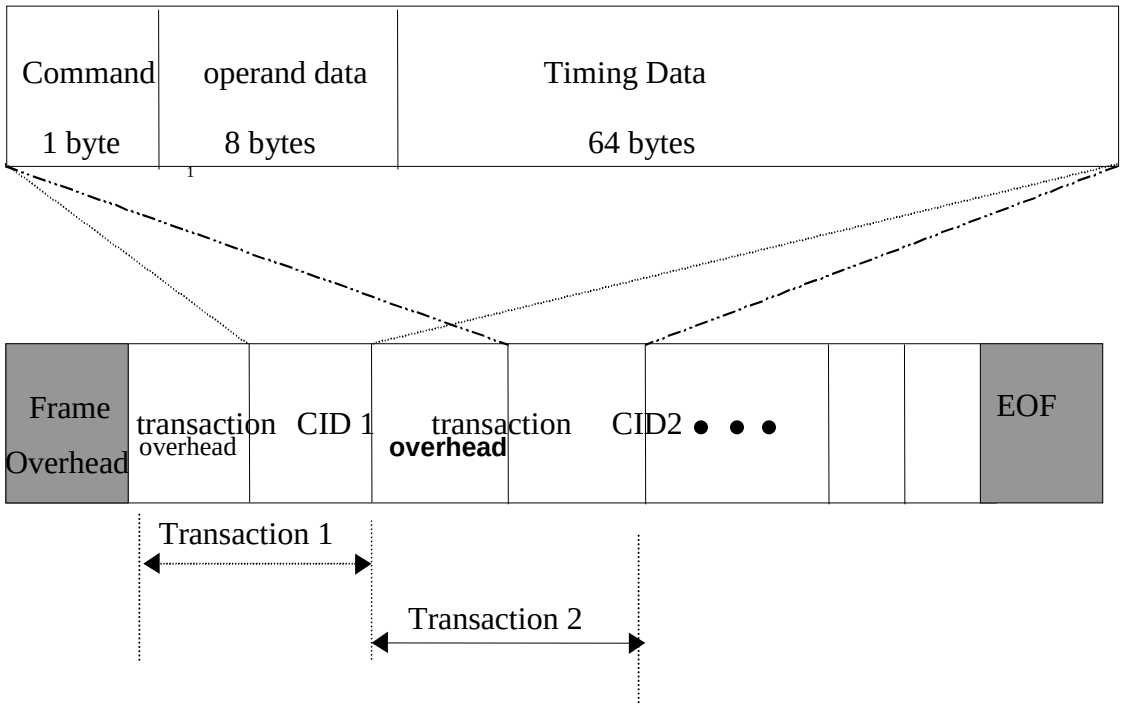

Figure 2.1. Computer to CID data format [1].

The computer is required to send data to the CID in 73-byte packets (Fig. 2.1). The first byte is a "command byte" provided to leave room for future features; in the CID II, it is always set to 1. The next eight bytes specify which of the CID's 64 outputs are to be turned on. The final 64 bytes specify pulse lengths in milliseconds for the first 32 outputs being turned on (two bytes each); a zero length causes the output to be left on indefinitely. The restriction of specifying pulse lengths for only 32 outputs is not particularly onerous in the CID's application.

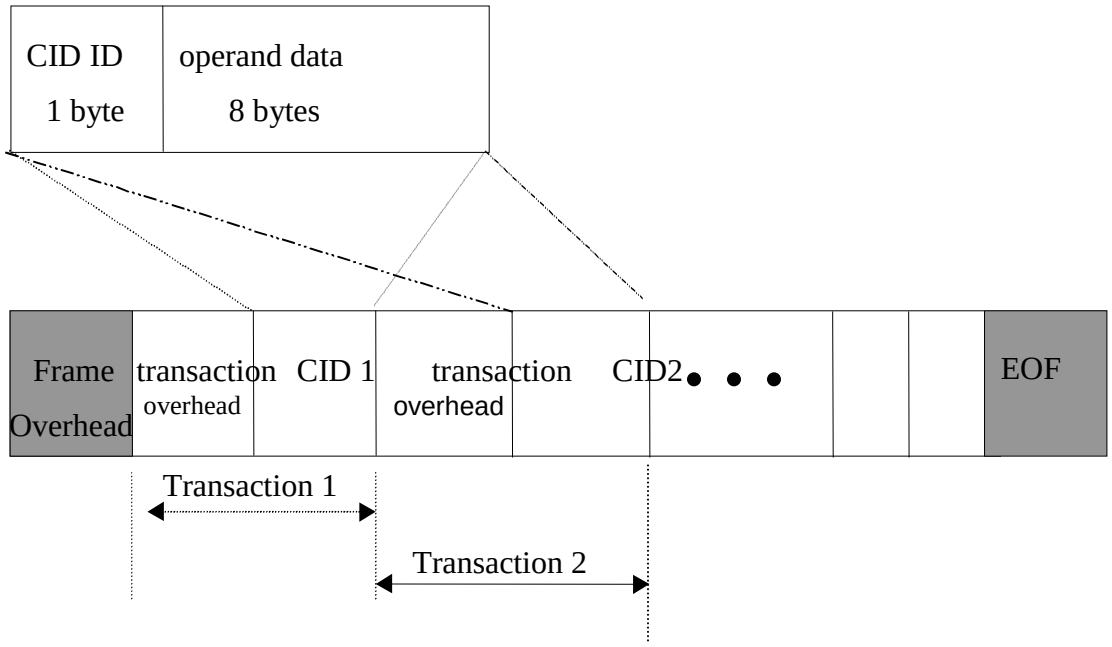

Figure 2.2. CID to computer data format [1].

The CID sends data to the computer in 9-byte packets (fig. 2.2). The first byte is a number uniquely identifying the CID; this number is set with DIP switches on the back of the CID's case. The remaining eight bytes give the present state of the 64 digital inputs.

While the computer only sends data to the CID once per simulation step, the CID sends data to the computer on every USB frame so that the state of its inputs (the traffic controller's phase outputs) is always available.

Chapter 3 – Real-Time Playback

Real-Time Playback (RTP) is a discrete-time simulation technique developed for systems in which it is difficult or impossible to "close-the-loop" between computer simulation and hardware testing—for instance, if the simulation is unable to be run in real-time. More generally, RTP can be used to create a quasi-hardware-in-the-loop simulation that is real-time to the limits of a playback device. Interaction with the physical system is independent of both simulation speed and communication latency between the simulator and the hardware.

3.1 Algorithm Description

RTP requires a playback device that can both *store* a series of inputs to be applied to the physical system and *record* the system's outputs.

The simulation procedure is as follows:

1. The initial state of the physical system's outputs is read from the playback device.

2. The computer simulator is started and run for a fixed amount of time, with the *simulated* system's output in the simulation fixed at its initial state. The simulation's inputs to the system are recorded.
3. The simulated system inputs are transmitted to the playback device, and "played" in real-time until a change is observed in the system's outputs.
4. The computer simulator is run again for a fixed amount of time past the previously observed output change. The output change is added to the simulated system's behavior. The simulation's inputs to the system are recorded.
5. Steps 3 and 4 are repeated, acquiring a new system output change each time, as long as desired.

Of course, the system must be in the same internal state at the beginning of each playback run.

3.2 Real-Time Playback in Power Systems

RTP simulators have been used by the electric power industry for some time. They provide a cheap alternative to Real-Time Digital Simulators (RTDS) in which a simulation actually interacts with the tested system in real-time. RTP simulators are used principally for testing numerical relays [7], but they have also been used for testing other types of hardware (for instance, fault locators [8]) with fast response times that preclude analog testing.

The greatest disadvantage of RTP simulations in power applications is simply the inconvenience of using a digital simulator for an analog system; care must be taken to minimize information loss due to quantization, conversion, and filtering [7]. A secondary disadvantage is that playback simulations are inherently slower than bona fide real-time simulations, requiring many runs of both simulation and playback, rather than a single one.

3.3 Application to Traffic Simulation

RTP can be used to create a pseudo-hardware-in-the-loop simulation that eliminates communication latency between the host computer and the CIDs. By comparing the results of RTP simulation to a normal hardware-in-the-loop simulation, the impact of latency on the simulation can be determined.

Because traffic controllers' inputs and outputs are digital, not analog, the first objection to RTP in power systems—that error might be introduced through quantization or conversion—is removed. However, the problem of increased simulation time is magnified by traffic systems' slow time constants; playback simulations in traffic systems will be extremely slow indeed. RTP is a tool suited best for verification, not for everyday use.

Chapter 4 – Building a Real-Time Playback Simulator for Traffic Systems

4.1 Overview

This chapter will discuss in detail the components of the RTP simulator for traffic systems.

The RTP simulator makes use of the existing CID II hardware, but with modified firmware. The VISSIM traffic simulator was used because of its capability of running with user-selected time step sizes, which allows testing for the impact of time step size as well as latency. A custom control program written in C++ manages the entire RTP process.

4.2 CID Modifications

At the hardware level, the CID II appeared capable of being turned into a playback device: the microcontroller is capable of timing outputs with 1 ms resolution, and has over 6 KB of on-chip RAM.

Three principle modifications were made to the CID firmware:

- Output timing data was removed from the design. The RTP simulation is designed to work with a 100 ms time step, and both the CORSIM and VISSIM simulators generate detector pulses that are multiples of 100 ms long.
- A queuing system was added, allowing the CID to store output data for a number of time steps in the future. This removed the possibility of error in the time step length.
- The CID was made to keep track of input changes. Specifically, the modified CID can record the value and time of the Nth input change after playback has begun (where N can be set by a command over USB).
- A four-byte simulation clock, incrementing every millisecond, was added to provide a timing base. Functions were added to enable multiple CIDs to synchronize their clocks prior to simulation.

The CID's output data queue is 64 time steps deep, where each time step is 100 ms long. The timing data bytes used in the CID II were eliminated, because CORSIM and VISSIM generate traffic detector pulses with 100 ms resolution.

The timing clock is four bytes long, but for convenience the least-significant byte rolls over at 100. This allows up to 4605.9 hours of playback before the clock rolls over.

4.2.1 CID Modes

A global "mode" byte allows the CID to keep track of its state from one start-of-frame interrupt to another.

In mode zero (the default startup mode), the CID acts more or less like a "normal" CID, sending its ID number and the status of its inputs over USB on each frame. It deviates from a normal CID's behavior in that timing data is ignored.

Modes one and two are used in conjunction with an external cable for simulation clock synchronization. In mode one, all 64 outputs are turned on whenever the least significant byte of the clock is zero. In mode two, the least significant byte of the clock is set to zero whenever any one of the 64 inputs is high. In both modes, the CID sends its ID number and the four bytes of its simulation clock over USB each frame.

In mode three, the CID waits until its clock reaches the simulation start time set beforehand, then *clears the clock* and enters mode four. It sends a zero byte over USB each frame; this allows the PC to determine when mode four has been reached.

Playback actually occurs in mode four. The CID increments the queue's tail pointer every 100 ms, and counts changes in the inputs. If the queue runs dry, the CID error flag is set to 1. When the Nth input state is reached, a flag is set and both the input state and the current time are recorded. (N is set by a command over USB.) In this mode, the CID sends nine bytes of data over USB each frame: its ID number (1 byte), the clock value (4 bytes), the number of bytes left in the queue (1 byte), whether the Nth input state has been reached (1 byte; either 1 or 0), a byte indicating whether the CID has been synchronized with another CID, and the error flag byte.

Modes five and six are used to report the Nth input state. In mode five, the CID sends its ID number (1 byte) and the stored time at which it detected the Nth state (4 bytes); in mode six, it sends its ID number and the actual values of the inputs at this state (8 bytes).

In mode seven, the CID sends its error flag byte to the computer on each frame. This is used by the control program to aid debugging.

4.2.2 CID Commands

The CID II had only one command: "set outputs." Additional commands were added to the playback CID to support the added functionality; Table 4.1 gives a summary of the commands.

Table 4.1. CID commands.

Command	Format
normal write	0x01 : output data (8 bytes)
queued write	0x02 : output data (8 bytes)
set start time	0x03 : start time (2 bytes – mid-significance)
set mode	0x04 : new mode (1 byte)
clear error byte	0x06
clear clock	0x07
set clock	0x08 : new clock value (4 bytes)

The *normal write* command is similar to the CID II write command (it sets the CID's outputs to the given values), but any timing data sent is ignored. It also empties the data queue and clears all simulation flags and variables except for the clock.

The *queued write* command adds the specified data to the output queue. If the queue is full, the CID error flag is set to 2.

The *set start time* command sets the clock value at which playback should begin. (However, the CID must be in mode 3 at the given time in order for playback to actually start.)

The *set mode* command puts the CID in the given mode. If the specified mode is outside of the acceptable range, the CID error flag is set to 3.

The *clear clock* and *clear error byte* commands clear the indicated variables. (However, there is generally no reason to clear the clock, and doing so in a multiple-CID environment will unsynchronize the CIDs.)

An invalid command byte causes the CID error flag to be set to 9.

4.2.3 CID Programming Issues

Two main issues were encountered in the process of making the necessary firmware modifications. They are noted here because they may point to problem areas for the CID II.

First, some difficulty was encountered in constraining the worst-case run time of the start-of-frame ISR (which contains most of the CID's functionality) to 1 ms. The biggest culprit was the one_ms_update() function, which handled output pulse timing. Converting this function to assembly and optimizing by hand seemed to mitigate the problem; and the function was later removed entirely when it was realized that 100 ms pulse resolution was sufficient. However, it seems possible based on this experience that worst-case ISR run time may exceed 1 ms in the CID II, causing possible pulse timing errors.

Second, a problem was found with the microcontroller development system's linker configuration. Certain variables in xdata (16-bit address) memory space were apparently located in invalid sections of memory, resulting in situations such as:

```
a = 5;
if (a != 5)  ErrorTrap();
```

where the example code *does* call ErrorTrap(). A temporary fix was achieved by using the compiler's _at_ directive to locate problematic variables in known good locations by hand. Due to time constraints, a better solution was not pursued, but future CID developers should be aware of the issue.

4.3 Playback Control Program

A control program, running on the simulation computer, handles the processes of running the traffic simulator, running the playback operation, passing data (phase states and detector pulses) back and forth, and determining when the simulation has completed or stalled. The program was written in C++ as a "Windows console program," meaning that it lacks a graphical interface but can still access Microsoft Windows API functions. The control program's operation can be divided loosely into three steps: initializing the simulation, running the playback loop, and generating results in a useful format.

4.3.1 Initialization

The control program performs the following initialization steps:

1. Read a configuration file specified on the command line.
2. Identify connected CIDs, and check that all CIDs required for the simulation are present.
3. Synchronize all CIDs' simulation clocks to insure that playback will start at the same time at every intersection.
4. Initialize traffic controllers by playing a predetermined sequence of data to each one to force it into a particular state.
5. Read each traffic controller's initial state.

The configuration file is a plain text file that specifies simulation parameters likely to change from one simulation to another. Figure 4.1 gives an example.

The CIDs are identified by scanning the system for CID devices, then reading data from each one to determine its ID number. If any CID listed in the configuration file is not found, the simulation is aborted.

```
# RTPB Configuration File.
# (Comments begin with a "#" sign.)

verbose           # Print ridiculously detailed messages.
overwrite         # Okay to overwrite old output files.

# Path to VISSIM executable.
vissim=C:\Progra~1\PTV_Vision\VISSIM360\Exe\Vissim.exe

# Traffic system definition file.
trafficsys=c:\rtpb\test.inp

# Files for passing phase and detector info to/from VISSIM.
phase=c:\rtpb\phase
detector=c:\rtpb\detector

# File in which to store the final list of phase and
# detector data.
output=c:\rtpb\output.txt

# ID numbers of the CIDs to be used in the simulation.
CID=5,6,7

# ID number of the CID to be used as the timing reference.
master=7

# Number of *additional* seconds to run VISSIM each time.
steptime=60

# Total run time (in seconds).
runtime=3600      # one hour
```

Figure 4.1. An example configuration file for the control program.

It is not strictly necessary to synchronize the CIDs' clocks for most simulations; however, it would be necessary for playback on advanced control systems with interaction between intersections. It also simplifies the process of managing data sent to the CIDs: when one CID needs more data sent, all the CIDs do. Synchronization is done one CID at a time by connecting the "master" CID to each "slave" CID in turn with an external "loopback" cable normally used for the CIDs' self-test function. The control program signals the master CID to turn its outputs on for 1 ms every 100 ms when the least-significant byte of its clock is reset to zero. The connected slave CID is ordered to clear the least-significant byte of its clock whenever any input is asserted. This synchronizes the least-significant byte, which changes too fast to be synchronized directly by the control program; the other bytes can easily be synchronized by the control program with the "set clock" command. The traffic controllers must be initialized before reading their "initial" state and at the beginning of every playback sequence in order to insure that they are in exactly the same state each time. This is done by asserting

detector signals to "call" the desired phases prior to starting the simulation. Ultimately, a more general and more customizable method of initializing the traffic controllers will be needed.
Finally, prior to beginning the first simulator run, the initial state of each traffic controller is recorded.

4.3.2 Running the Traffic Simulator

In the control program, a C++ class is instantiated for each CID. The class provides methods for sending and receiving data, "shortcut" functions for common CID commands, and data storage arrays for both phase and traffic detector data. When the associated traffic controller's initial phase state is read, it is copied through the CID's entire phase storage array. Then a member function is called to dump the array to a file to be read by VISSIM.
The length of time to run VISSIM is initially set to the "steptime" value specified in the configuration file, and then the VISSIM executable is called with the appropriate command-line arguments.

4.3.3 Running the Playback Operation

When VISSIM finishes its simulation run, a class member function is called for each CID to load its own newly-generated traffic detector data file into its detector data array. A block of detector data is then sent to each CID to be loaded into its data queue. Each CID is instructed to look for its first phase change. The simulation clock is read from one of the CIDs, and the playback start time is set for a few seconds in the future. Once playback starts, the control program runs in a loop, checking the amount of data left in the CIDs' queues and sending more data when necessary. Playback ends when one of the CIDs reports a new phase change. (Simultaneous or near-simultaneous phase changes are resolved intelligently.) The value and time of the new phase state are added to the CID's phase array, and the number of the phase that CID is looking for is incremented: on the second playback run, one of the CIDs will be looking for its *second* phase change.
Each CID's phase data is again dumped to a file, the VISSIM run time is updated, and a new simulation cycle begins. The process continues until the number of simulation-seconds specified in the configuration file have been simulated.
At the end of the simulation, all confirmed phase and detector results are written to a text file.

4.4 Simulator Interface

The interface to the computer simulator must allow traffic detector pulses to be recorded for and phase states inserted by an external program. Fortunately, these are the same functions that were already required in order to interact with CIDs in a real-time simulation.

Although most of the CID research so far has been conducted with TSIS/CORSIM, the VISSIM simulation package was chosen for use with this research, because the latest version allows the time step size to be set arbitrarily.

Zhen Li, a Ph.D. student in civil engineering at the University of Idaho, developed both the CORSIM and VISSIM interfaces to the CID II, and was kind enough to adapt his VISSIM interface code for the playback system.

4.4.1 Sharing Data with VISSIM

On startup, VISSIM spawns an interface program for each CID. The interface program provides two function hooks of interest: one called before simulation starts, the other after simulation is completed.

The first function loads one hour (with 100 ms time steps) of phase data for the given CID from a file called "phaseN", where N is the CID's ID number. The file contains the contents of the corresponding phase array in the control program. The second function dumps the contents of the detector array for the intersection into a file called "detectorN", where N is, again, the CID's ID number.

Originally, the files were human-readable text files; later, in an effort to debug by simplifying, they were turned into simple byte-dumps of the detector data arrays.

4.4.2 Setting the Simulation Run Time

The VISSIM simulation setup file is a human-readable text file, making automatic configuration changes easy. The amount of time to run the simulation is set by the control program, which simply searches for and replaces the appropriate line in the setup file, specifying the desired new run time. The simulator is run for "`timeCompleted + simsettings.steptime`": the amount of time already completed by the playback simulator plus an additional length of time specified in the playback simulation settings file.

4.4.3 Running VISSIM

VISSIM provides a command-line interface that allows the program to be started with a given simulation file. It runs the simulation, including the adapted CID interfaces, and exits automatically.

In order to tell when VISSIM has completed running, the control program deletes the (newly obsolete) detector file for one of the CIDs prior to starting VISSIM, and then checks repeatedly for the file's existence. (The detector data files are created at the end of the simulation run.)

Chapter 5 – Real-Time Playback Testing

Two tests were desired to verify the correct operation of the real-time playback simulator. First, a two-phase, single-intersection simulation was used for ongoing testing during development, the goal being to achieve a single-intersection playback simulation. Second, a multiple-phase, multiple-intersection simulation was to be used to test multiple-CID operation.

Late complications with the first simulation brought to light additional features necessary to complete the playback simulation program, but also prevented work from progressing to the multiple-intersection simulation; thus, the multiple-intersection functionality remains untested.

The single-intersection simulation was made to work, except that repeatability problems were discovered. An underlying assumption of the RTP system is that, for instance, the 5^{th} phase change always occurs at exactly the same time in the playback sequence. This turned out to not be the case, probably because the existing procedure for resetting the traffic controller is not sufficient.

A new procedure has been devised for resetting the traffic controllers. For example, suppose an intersection has only two phases. The first phase is called for a "long time" to insure that the traffic controller has activated that phase. Then the second phase is called and CID outputs are turned off for a set period of time. This sequence is designed to not only place the traffic controller in a particular phase state, but to insure that it has been in that phase state for the same length of time each time that playback is begun.

More generally, a predetermined sequence of inputs must be applied to each traffic controller to insure that it starts the simulation in the same state each time. Both this problem and its solution were predicted in [9]. Implementing this solution requires firmware modifications, because the CID firmware is set up to look for the (N)th phase change, and there's no way of knowing how many phase changes will occur in the reset sequence. The obvious way of fixing this is to make the firmware look for the (N)th phase change "after the simulation clock is > 30 seconds" (or however long the reset sequence lasts).

Chapter 6 – Summary & Topics for Future Research

A procedure has been developed to evaluate the accuracy of hardware-in-the-loop simulation for a given traffic system by using the real-time playback technique. Additionally, the procedure can be used in conjunction with a variable time-step simulator to evaluate the affect of time-step size (either in addition to or independent of communication latency).

A number of possibilities for future research exist. First, the critical modification to the RTP simulator suggested in Chapter 5 should be completed—namely, improving the traffic controller reset function—and the simulator's operation should be verified with a multiple-intersection system. Second, the completed tool should be used in evaluating hardware-in-the-loop simulation for a representative set of traffic systems: a step initially

planned for this thesis, but removed from the scope by the necessity of expedient graduation. It would be useful, in particular, to test a simulation with a large number (30-50) of CID-run intersections, in order to either find an upper bound on simulation size or guarantee the accuracy of such a simulation. This is particularly of interest in light of possible CID II orders from traffic planners hoping to run simulations approaching this size.

The CID clock synchronization process is cumbersome, and could be removed altogether for the majority of simulations. If this were done, it might be worthwhile to multithread the portion of the program that feeds data to the CIDs during playback, so that each CID had its own thread.

Many improvements are still possible to the real-time playback software, particularly in the "control program," which was under development until the last possible minute. Most of these are cosmetic, making the software suitable for more general use, or to add robustness. An exhaustive list of possible changes and additions is given in Appendix F.

More research is also needed into methods of improving the CID latency problem. This will become particularly important if major traffic simulation packages move towards a smaller (100 ms) time step, as has been suggested: the maximum number of CID-controlled intersections will become severely constrained by the necessity of communicating with all CIDs within one time step. Three main approaches are possible:

1. Use different USB transfer methods, such as "bulk" transfer mode. Testing of actual communication time would be important: in the existing CID, the excessive delay in driver accesses was not foreseen; it had to be observed in the lab.
2. Modify the USB driver to support faster isochronous accesses, if possible. Unfortunately, few individuals have driver development experience, even in a university setting.
3. Employ software tricks to make the existing driver and hardware work fast enough: for instance, one possibility would be to access multiple CIDs from different threads.

References

1. Ying Zhou, "Real-Time Traffic Simulation," MSEE thesis, University of Idaho, 2000.
2. [re-tracking down this reference].
3. "Hardware-In-The-Loop Traffic Simulation," Texas A&M University, http://translink.tamu.edu/docs/Labs/Hardware-in-the-loop/HWLSPage.pdf.
4. Bullock, D. and A. Catarella, "A Real-Time Simulation Environment for Evaluating Traffic Signal Systems," Transportation Research Record, #1634, TRB, National Research Council, Washington, DC, pp. 130-135,1998.
5. D. Bullock, private email, February 4, 2002.
6. R. B. Wells, J. Fisher, Y. Zhou, B. Johnson, M. Kyte, "Hardware and Software Considerations for Implementing Hardware in the Loop Traffic Simulation," IEEE IECON'01, November 2001.

7. M.S. Sachdev, T.S. Sidhu, P.G. McLaren, “Issues and Opportunities for Testing Numerical Relays,” IEEE Power Engineering Society Summer Meeting, July 2000.
8. R. Das, M.S. Sachdev, T.S. Tidhu, “A Fault Locator for Radial Subtransmission and Distribution Lines,” IEEE Power Engineering Society Summer Meeting, July 2000.
9. R. Engelbrecht, “Using Hardware-in-the-Loop Traffic Simulation to Evaluate Traffic Signal Controller Features,” IEEE IECON’01, November 2001.

Other references:

- D. Bullock, B. Johnson, R. Wells, M. Kyte, and Z. Li, "Evaluation Procedures for Traffic Signal Equipment using Hardware in the Loop Simulation," Transportation Research Part C, Pergamon Press, submitted April 2001.\
- Universal Serial Bus Specification, Revision 1.1 September 23, 1998. http://developer.intel.com/technology/usb/spec.htm.

Part 4: Development of a USB for the CID with Reduced Latency

Chapter 1 Introduction to the Controller Interface Device (CID II)

1.1 Real World Traffic System

In our everyday life, we pass many signalized traffic intersections, but many of us don't know that there is complex logic controlling the traffic signals. These traffic signals are controlled by traffic controllers, which are programmed using different algorithms. The complexity of these algorithms depends upon factors such as the number of vehicles, time of peak traffic, etc. But in general these algorithms implement either a fixed-time scheduling system or a traffic-actuated control system.

In a fixed-time scheduling system, the traffic controller allows the flow of traffic in one direction for a predetermined amount of time and then switches the direction. The fixed-time scheduling systems are more suitable for places that have heavy traffic or many intersections close together. But in many cases, a traffic-actuated control system where the traffic signals respond to the present traffic in a more efficient method. The real time traffic data is obtained using loop detectors or video cameras.

1.2 Simulating the Real World Traffic

A traffic simulation program helps traffic engineers to analyze the real world traffic when control algorithm changes are introduced without being at the risk of disrupting the traffic. There are several traffic simulation programs available that model the behavior of real-world traffic. CORSIM is one such simulation tool, developed by the Federal Highway Administration (FHWA) [2]. VISSIM is another simulator developed by Innovative Transportation Concepts, Inc [12]. These simulators help users to determine the measures of effectiveness (MOE) for traffic systems, which include vehicle delay, stopped delay, queue lengths, and other detailed run results.

1.3 Hardware-in-the-Loop Simulation

Since there are several simulators from different vendors, it is difficult to choose between these simulators. For this analysis, we use a common simulator to simulate the traffic and then we interface with a real-time traffic controller using a CID. On each time step, phase data from the traffic controllers are recorded and transmitted to the simulation. Then traffic detector pulses generated by the simulation program are sent back to the traffic controllers. The diagram in Figure 1.1 illustrates the concept.

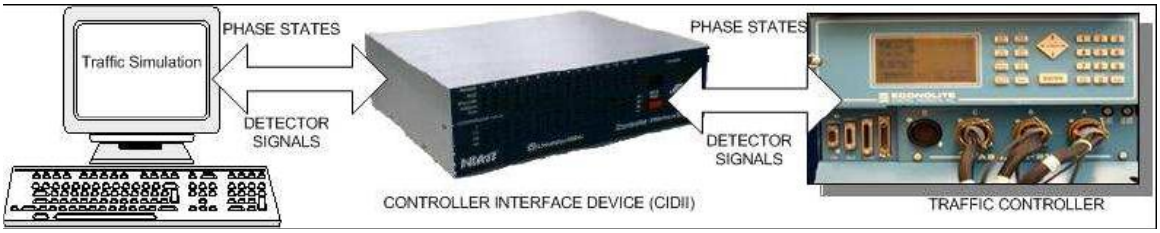

Figure 1.1: Hardware-in-loop simulation

1.4 Concept of the original controller interface device

The first CID was developed at the Louisiana State University in 1997 for the Federal Highway Administration under contract from ITT Systems and Sciences Corporation [7]. Concurrently another prototype CID was designed at the TransLink Research Center at the Texas Transportation Institute for use in its Roadside Equipment Laboratory (REL) [1]. The former used a serial communication interface while the latter used a computer parallel port as a communication interface. Although the concept of the CID got attention from many people, it had certain drawbacks. The first limitation is that the number of CIDs that can be connected to the simulation models for real-time simulation is limited, limiting the number of intersections one can simulate using CIDs to roughly 16. This limit is imposed by the simulation time step and communication latency introduced by the use of serial or parallel interface. Finally the detector signal had no timing parameter, which is needed to represent the length of the detector actuation.

In 2002, the University of Idaho developed the second generation Controller Interface Device (CID II), which was an improvement over the previous version of CID. The CID II uses USB 1.1 interface as the communication interface between the computer and CID. This interface was capable of sending data at a much faster rate than a serial or a parallel interface used by the previous generation CID, allowing more intersections with controllers and CIDs in a single simulation. In addition, more data per traffic controller can be sent. Theoretically the USB interface allows up to 127 traffic controllers to be connected in one simulation.

1.5 Data format for the Controller Interface Device (CID II)

This section explains the format of data that is sent to and from the CID. The CID is treated like any other USB device by the operating system. The host has to initiate all of the data transfers. The firmware for the CID defines two data formats: one for data sent from computer to the CID and the other for the data sent from the CID to the computer. Figure 1.2 shows the data format of data sent from the computer to the CID. The computer sends 73 bytes of data to the CID in a packet. The command byte is always set to 1. The operand data specifies the state of the 64 outputs indicated by the LEDs on the CID. The 64-byte detector actuation timing data is used to store the timing for the first 32 detectors actuated. The timing data for each detector actuation is 2 bytes in length. The data from the CID to the computer are 9 bytes in length. Figure 1.3 shows

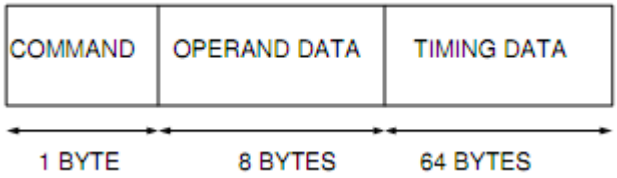

Figure 1.2: Computer to CID data format

the data format. The DIP switches set the value for the CID ID. The operand data represents the 64 digital input bits.

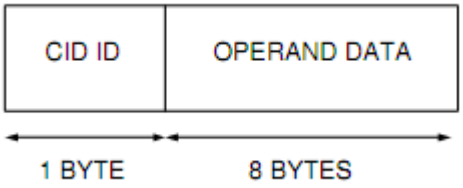

Figure 1.3: CID to Computer data format

1.6 Problems encountered with CID II

The CID uses the USB 1.1 interface for communication with the host computer. The method of transfer employed in the CID II is the isochronous transfer mode, which can transfer up to 1023 bytes of data in a 1-millisecond (ms) frame, with a guaranteed update rate of 1 ms. As noted above, 73 bytes are written and 9 bytes are read for each CID, hence it should be possible to include more than 1 CID's data in a single USB frame. However the current windows USB driver requires 6 ms to execute a read from each CID and another 6 ms to execute a write. The procedure employed to determine the 6 ms read/write delay is described in Appendix A. Also the current driver does not support multiple simultaneous-isochronous mode transfers. This delay is a hindrance to the performance of hardware-in-the-loop simulations. The current simulation time step is 1000 ms, so the delay is not significant if 40 CIDs are included [5]. But there are plans to reduce the time step to 100 ms in some simulation programs. If that happens, then this delay limits the number of CIDs that can be included in one simulation study significantly.

1.7 Designing a Custom Device Driver

This thesis will document the process of designing, implementing and testing a Windows device driver for the CID II. A device driver is a piece of software that provides the communication between the operating system and the hardware. The driver will be a Windows Driver Model (WDM) driver. It is advantageous to write a WDM driver as it provides binary compatibility between Windows 98, Windows 2000, and Windows XP. This

binary compatibility allows the CID II to function on all of these operating systems without the need to change the driver code.

1.8 Overview of the Thesis

Chapter 2 provides information about Windows device drivers. All the important data structures used in the driver are also explained in Chapter 2. Chapter 3 gives an introduction to the USB interface used for this project. Chapter 4 describes the new CID II device driver and describes all the important code fragments of the device driver. Installing the device driver is explained in Chapter 5, which also includes a sample INF file. Building, testing and debugging techniques are explained in Chapter 6. Chapter 7 presents the results obtained using the new device driver for the CID II. This thesis concludes with Chapter 8, which provides a brief summary and insight into future research possibilities.

Chapter 2 Introduction to Writing Windows Drivers

It is always important to understand the internal functions of any operating system before starting to write a device driver. This chapter provides a brief introduction to the Microsoft Windows architecture. However this chapter only emphasizes information that is closely related to driver development.

2.1 Overview of Windows Architecture

Before describing the driver itself, it helps to understand some of the terms commonly used while describing internals of Windows operating system.

Win32 API functions: These functions are the normally used to write any Win32 applications. Examples include the CreateProcess and the CreateFile system services. These services are invoked from the user mode. For example, NtCreateProcess is the internal system service that is called when a Win32 API call for CreateProcess is invoked.

Kernel support functions: These are functions that exist inside the kernel of the Windows operating system. For example, KeInitializeEvent is used to initialize an event and wait until the completion of that event. Win32 services processes started by the Windows 2000 service control manager.

DLL (dynamic-link library): A DLL is a library of executable functions used by the 32 bit Windows applications. A DLL provides one or more particular functions; Windows API programs create a dynamic link to the DLL, for example: Msvcrt.dll, Kernel32.dll etc.

The diagram in Figure 2.1 shows how various program layers interact in the Windows operating system. Consider an application in the user mode, which makes a Win32 API call to the Win32 subsystem. The I/O Manager receives the call first and then passes this message to the appropriate device driver to handle the request made by the Win32 program. The unit of information passed from I/O manager to the device driver is known as an I/O Request Packet (IRP). The device drivers send the required information to the hardware through the hardware abstraction layer (HAL). The HAL essentially isolates the hardware from the burden of operating system specifications.

2.2 Different types of Drivers

The driver organization in Windows 2000 is shown in Figure 2.2 [8]. The kernel mode drivers are classified based on their function.

- **File system drivers**: These are the highest level drivers, and they include system supplied FAT16, FAT32, NTFS file system drivers.
- **Legacy drivers**: These are the lowest level drivers, and they directly control the I/O bus. They don't depend upon any other drivers nor do they have any kind of abstraction.
- **PnP**: All Plug-n-Play drivers have automatic resource allocation done by the Plug-n-Play manager. The PnP drivers provide a very good level of abstraction for driver writers. A subset of the PnP driver is the WDM. The driver written for the CID II is a WDM driver.

- **Video drivers**: These drivers manage the graphics display for the PC. They are usually shipped by the graphics chip manufacturers. There are also general purpose video drivers provided by Microsoft Corporation with the Windows operating system.

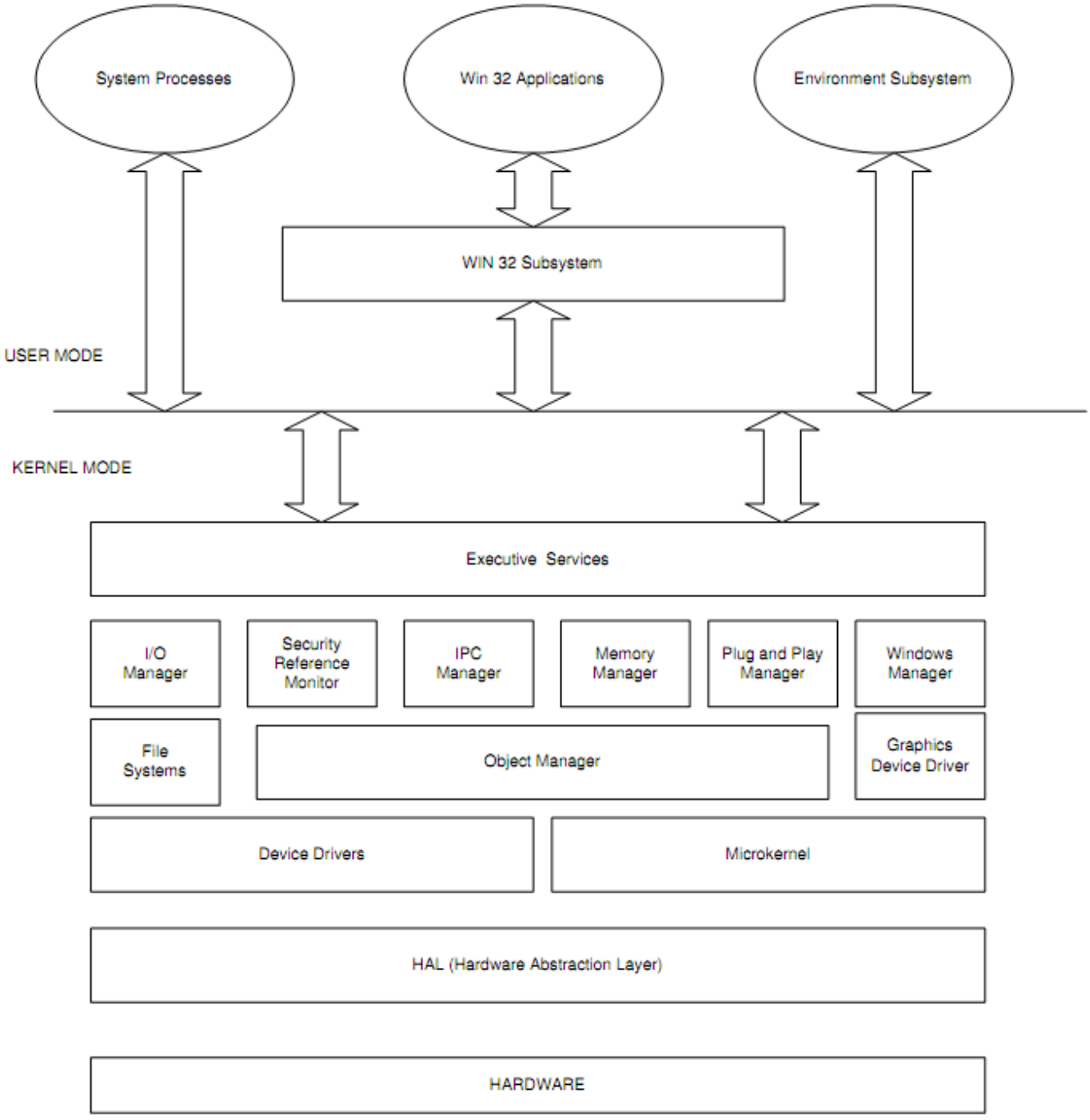

Figure 2.1: Architecture of the Windows 2000 operating system.

Figure 2.2: Classification of drivers.

2.3 Why Write a Windows Driver Model (WDM) Driver?

There are several reasons one would want to write a WDM driver:

- The hardware designed does not have any drivers for the operating system.
- The pre-existing device driver does not perform a required function on a hardware device adequately.
- To better understand the internals of Windows operating system.

It is always best for the driver developer to write a WDM driver. Microsoft specifies that a WDM driver is binary compatible, which means that the driver once coded will work across various Windows operating systems for the same hardware.

2.3.1 Classification of WDM drivers

There are basically three types of WDM drivers. Figure 2.3 explains the relationship between the drivers.

- **Bus Driver**: The bus driver enumerates the devices attached to the bus and creates Physical Device Object (PDO) for every device. The bus drivers are usually shipped with the operating system. The USB driver (USBD) is an example of a bus driver.
- **Function Driver**: A function driver is usually provided by the device vendor. The Microsoft Windows operating system also includes some functional drivers for the commonly used hardware. The driver designed for the CID II is a function driver.
- **Filter Driver**: This driver enables the use of any new features provided by the vendor for their hardware. The filter drivers have extended functionality that can be either a lower level filter driver or an upper level filter driver. A lower-level filter driver modifies the behavior of the hardware device, while a higher-level filter driver adds features for a device.

2.4 Data Structures in the Device Drivers

The development of a device driver requires the use of several data structures. These structures are normally defined in the header file of the driver source code.

2.4.1 Device Object

Device objects are data structures that help the operating system recognize the hardware. Every level of WDM drivers have their own device object. For example the functional drivers create a Functional Device Object (FDO) and the bus drivers create a Physical Device Object (PDO). The diagram in Figure 2.4 is the data structure for a device object used in CID II driver.

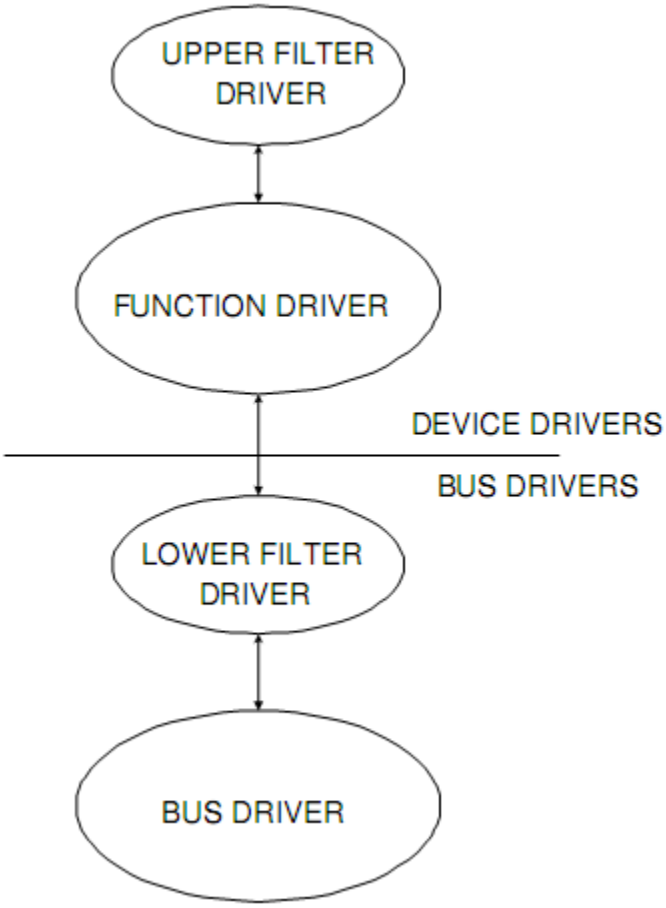

Figure 2.3: Classification of WDM drivers.

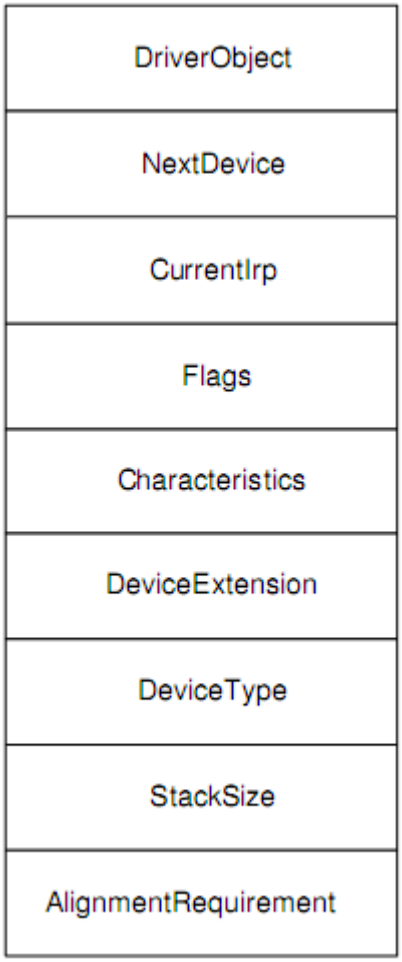

Figure 2.4: Device Object Data Structure

PDRIVER_OBJECT DriverObject: The *DriverObject* is a pointer to another data structure, representing the driver's loaded image that was input to the DriverEntry and AddDevice routines.

***PDEVICE_OBJECT* NextDevice**: Points to the next device object in the stack of device objects if any, created by the same driver. In a WDM driver, the I/O Manager takes care of this stack after each successful call to *IoCreateDevice()*. A Non-WDM driver however must do the job of unloading and deleting the list of its device objects.

PIRP CurrentIrp: Points to the current IRP sent to the *StartIO* routine.

ULONG Flags: The driver clears the flag in the AddDevice routine and this clearing is performed by ORing the field with one of the power flags (these are set after the device is powered up). The flags that are used in the driver are:

- DO_BUFFERED_IO or DO_DIRECT_IO: This flag must be set to perform direct I/O requests.
- DO_BUS_ENUMERATED_DEVICE: The drivers will not modify this flag, but the system itself sets this flag in each PDO.

- DO_DEVICE _NITIALIZING: The I/O Manager sets this flag when it creates the device object.
- DO_POWER_PAGABLE All the pageable drivers (Windows 98, Windows Me, Windows 2000, Windows XP) set this flag.

ULONG Characteristics: Another set of flags usually used by storage devices

PVOID DeviceExtension: It points to the device extension data structure that is created in the header file of the driver.

DEVICE_TYPE DeviceType: This describes the type of device that is attached. A more detailed list of devices are defined in wdm.h and ntddk.h.

CCHAR StackSize: Contains the count of stack locations in IRPs to be sent to the driver. The DDK routines *IoCreateDevice()* and *IoCreateDeviceSecure()* set this field to "1" in newly created device objects. The lowest-level drivers can therefore ignore this field.

ULONG AlignmentRequirement: When performing data transfers it is important to know the device's address alignment. Again this value is defined in wdm.h and ntddk.h.

2.4.2 Driver Object

Another important data structure in the driver is the driver object. The driver object represents an image of a device driver. Some fields in the driver object are opaque and are not accessible by the user. The accessible members of the driver object are

DEVICE_OBJECT DeviceObject: A pointer to the device objects discussed earlier. A call to *IoCreateDevice()* routine updates this member.

PDRIVER_EXTENSION DriverExtension: a pointer to the driver extension data structure. The driver extension is another data structure with ***AddDevice()*** as the only member that is accessible.

PUNICODE_STRING HardwareDatabase: Points to the \Registry\Machine \Hardware path in the hardware configuration information in the registry.

PFAST_IO_DISPATCH FastIoDispatch: File storage devices make use of this pointer, which points to driver's fast I/O entry points.

PDRIVER_INITIALIZE DriverInit: The entry point for the DriverEntry routine.

PDRIVER_STARTIO DriverStartIo: This points to the entry point for the driver's StartIo routine.

PDRIVER_UNLOAD DriverUnload: The entry point for the driver's Unload routine.

PDRIVER_DISPATCH MajorFunction: This has the IRP major function codes like IRP_MJ_CREATE, IRP_MJ_PNP, IRP_MJ_POWER, IRP_MJ_READ and IRP_MJ_WRITE.

2.5 Interrupt Request Level

The Interrupt Request Level (IRQL) is a method used by the Windows operating system to synchronize between different tasks. A process of a certain IRQL can be stalled only by another process with a greater IRQL. Most of the user mode programs execute at an IRQL known as the PASSIVE_LEVEL.

All the interrupt levels used in the CID are shown in Figure 2.5, where the lowest number represents a lowest priority. All hardware devices are assigned an interrupt request level known as the Device IRQL (DIRQL). This value is passed to the driver by the Plug and Play (PnP) Manager. The APC_LEVEL is used by the system to execute Asynchronous Procedure Calls (APC).

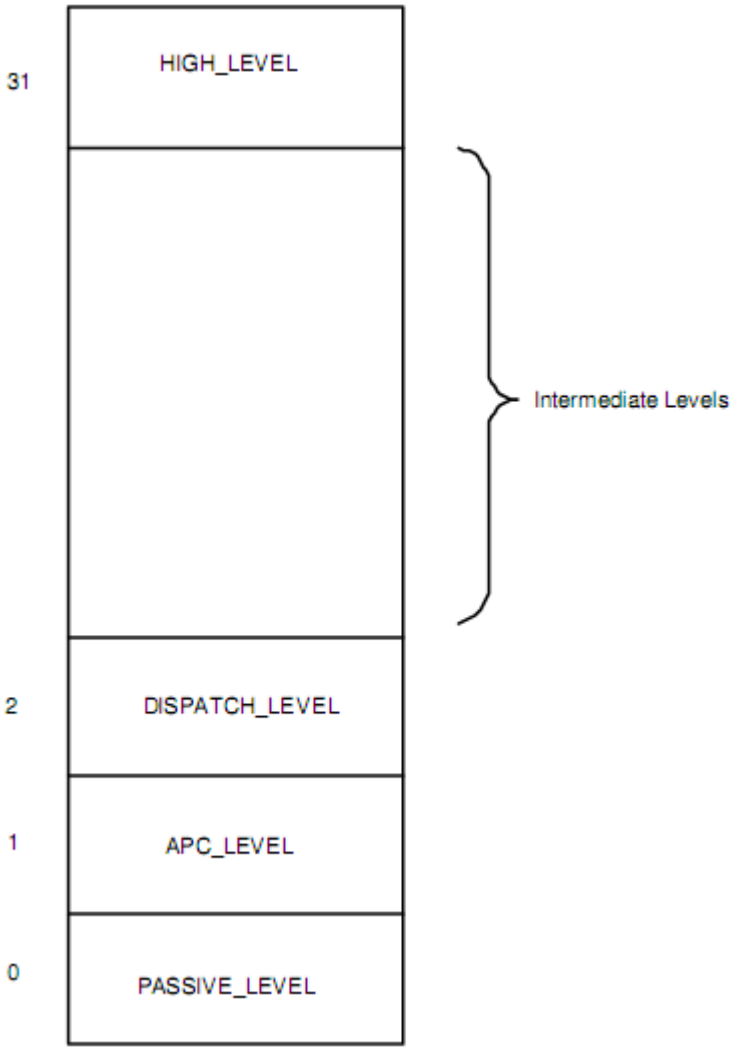

Figure 2.5: Interrupt Request Levels.

2.5.1 Synchronization using IRQLs

Access to a data object must be done at a level greater than the PASSIVE_LEVEL, which essentially prevents preemption by any other applications. Any code that is executed at DISPATCH_LEVEL or greater must not

generate page faults. This can be achieved by allocating non-paged memory. But the non-paged memory is a limited resource, so the user should be careful to free the memory after using it.

2.5.2 Spin Locks

In a multiprocessor, computer IRQLs cannot really protect data integrity. The use of spin locks provide a method to lock the code until a CPU is done reading or writing data. A drawback in using spin locks is that no other task can be done when the CPU is waiting to acquire a spin lock. For a code to obtain a spin lock, it must be executing at or below DISPATCH_LEVEL.

2.6 Summary

This chapter provides the basic information required to start writing a device driver. It is very important to refer to the most recent Driver Development Kit (DDK) for all the implementation specifications.

Chapter 3 Communications using the Universal Serial Bus (USB)

3.1 Origin and Classification of USB

Computer input/output interfaces are undergoing a rapid transition and only the interfaces with enough good features can succeed in today's competitive market. USB (Universal Serial Bus) is one such computer interface. Some of the important features of USB are

- Inexpensive and efficient
- Plug-n-Play, making it user friendly
- It is easy to design USB based devices
- Allows up to 127 devices, possibly of different types from different vendors to be connected to the PC

USB devices, once plugged to the PC, are automatically configured and the user need not worry about changing any IRQ settings or other hardware settings. Most newer PC's come with at least two USB ports. A USB hub can be used to increase the number of USB ports connected to one computer. The USB architecture supports three different transfer speeds; the maximum transfer rate is 480 Megabits per second, the lowest speed is 1.5 Megabits per second. There is also an intermediate speed of 12 Megabits per second. The different speeds benefit a wide range of users. The system designers can now design low-speed, inexpensive USB devices without much complexity in design. USB provides users with 3 different speeds

(1) Low Speed Devices (1.5Mb/s): These are devices that need to be manufactured at a lower cost, but still be able to transfer data at a considerable speed. The cables and connectors for the low speed devices are available at low-cost. Examples: Keyboards, Gaming Peripherals

(2) Full Speed Devices (12Mb/s): Devices that are required to have good communication speed use this transfer method. Examples: Integrated Services Digital Network (ISDN), Private Branch Exchange (PBX)

(3) High Speed Devices (480Mb/s): This speed was introduced in USB 2.0 Video players and streaming music players are examples of high speed devices.

3.2 USB Bus Topology

The PC is the host in the USB system, with a host controller as an interface. A root hub is also associated with the host system.

Figure 3.1 represents the bus topology of a USB system. The USB system can have a maximum of one host. The root hub has one or more USB ports. A single USB device connects to a USB port. A hub can be used to add multiple USB devices.

3.3 Communications in the USB system

In simple terms, all of the data is exchanged between the host and the device, with the aid of system software. The Microsoft Windows operating systems used here have built-in support for USB. But installing some types

of USB devices requires a device driver. This required device driver or the system software is usually provided by the hardware manufacturer.

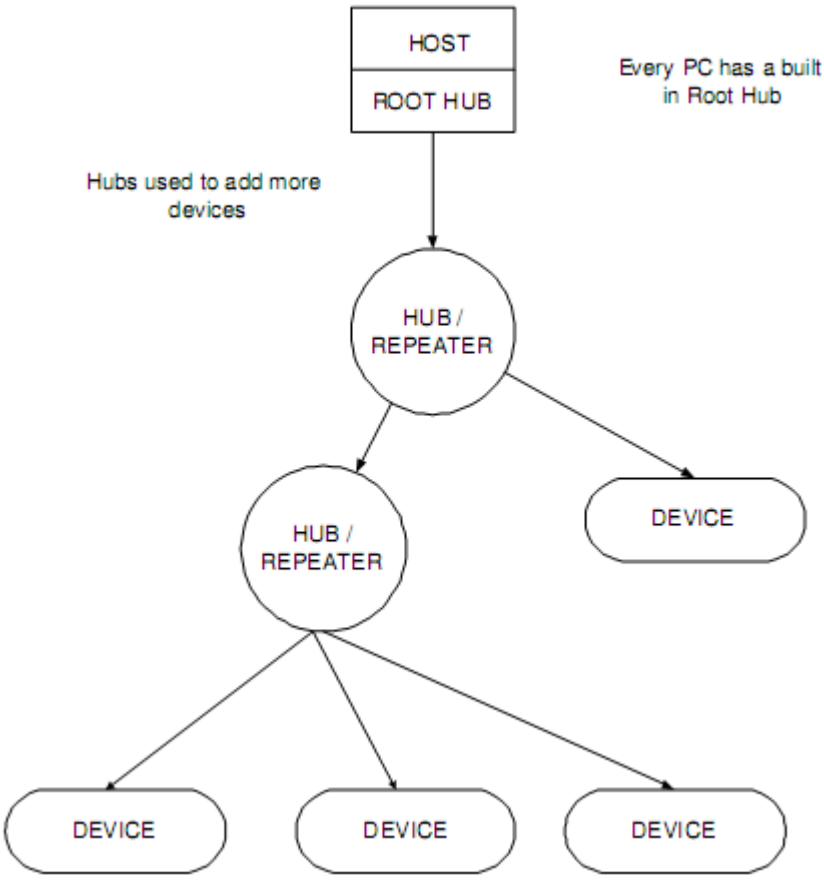

Figure 3.1: Physical Connectivity of the USB System

The interaction among various components of the USB system is shown in Figure 3.2.

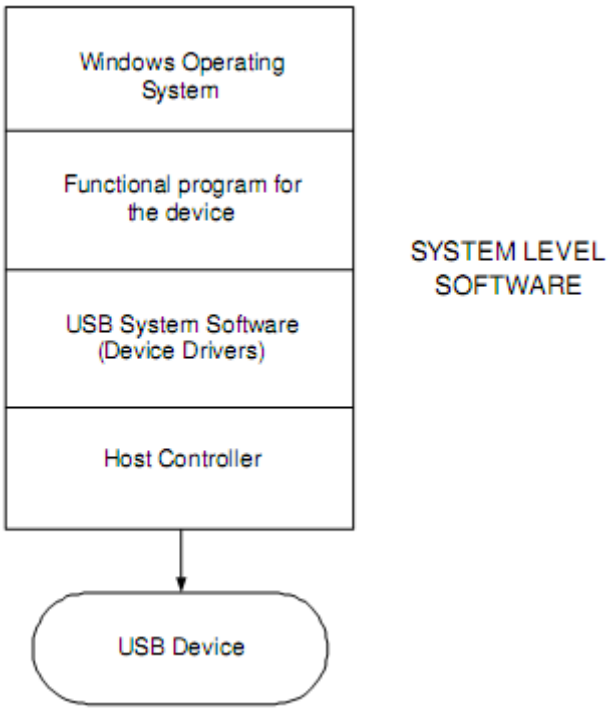

Figure 3.2: Communications in the USB

Starting at the bottom in Figure 3.2, there is a USB based device. The host controller is located in the PC, this also includes the bus driver for the USB. The USB system software includes the device drivers for different USB devices. The client software designed for the USB device is in the functional layer.

A more detailed working knowledge of USB communications is presented in Section 3.3.1. The host is the master and controls all of the transfers in a USB system. There are no peer-to-peer communications allowed in a USB system. Figure 3.3 shows the interaction between the various components involved in a USB transfer.

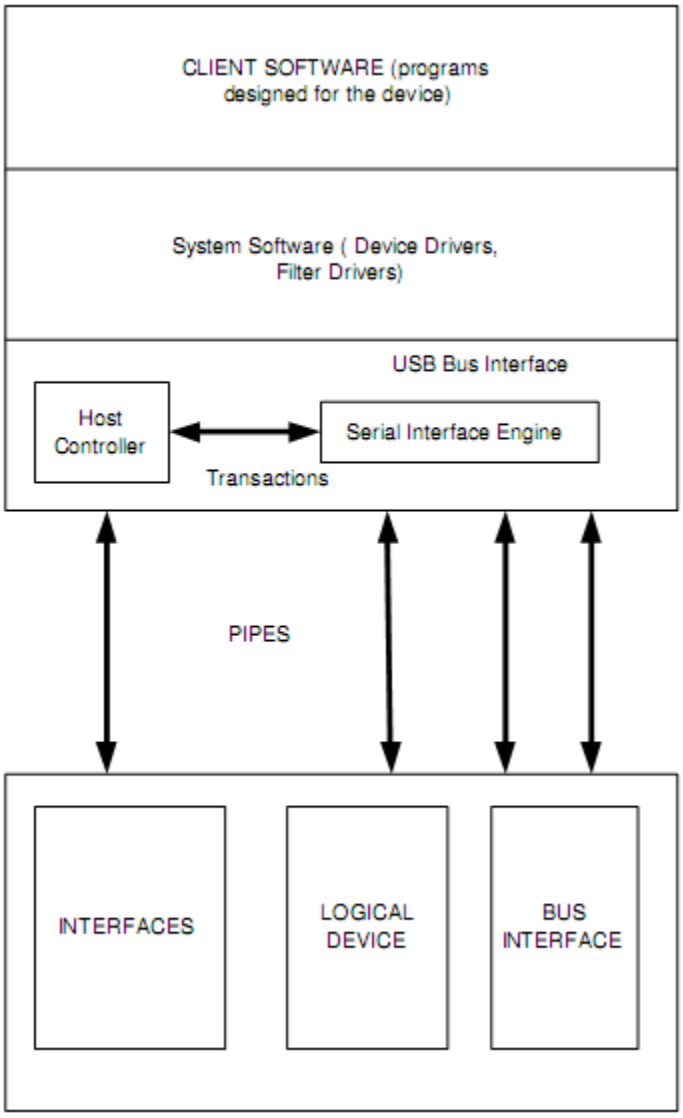

Figure 3.3: Host and the Device Connectivity

3.3.1 USB Communication Components

(1) Host Controller Driver (HCD): The Host Controller Driver is an operating system independent implementation for the USB system. It helps in allocation and deallocation of Host Controller resources. The software interface that provides this abstraction is known as the Host Controller Driver Interface (HCDI).

(2) USB Bus Driver (USBD): The USB bus driver is a low level driver, usually shipped with the operating system. USBD driver passes the requests from the function driver to the device.

(3) Endpoints: Endpoints are buffers present at the device end of the pipes. Endpoints support data to travel in either direction. Endpoints determine the bandwidth requirements and the packet size that can be transmitted or received. Endpoints are also responsible for the transfer type and direction of the USB transfers.

(4) Pipes: Pipes are the logical bridges between the host and the endpoints on the device. There are basically 2 types of pipes

(a) Message Pipes: These are the bidirectional pipes, normally used for the control transfers during the USB initialization process. It is important for the data to be structured when using a message pipe. The message pipe cannot have a different endpoint number for each direction [3].

(b) Stream Pipes: These pipes transfer data only in one direction. There is no required structure for the data to be sent across the stream pipes.

3.4 USB Transfer Terminology

To have a clear understanding of the USB communication, it is important to understand various terms used to describe USB transfers. Figure 3.4 represents various transfers, transactions and packets involved in USB communications. Every USB transfer consists of one or more transactions, and every transaction or stage will contain a token, optional data and an optional handshake. Every packet or phase contains a packet identifier (PID) and sometimes the packet contains additional information and error checking bits.

All transactions have a setup phase, this setup phase requests information about the USB device. All USB transactions are defined from the host perspective, so an IN transaction means data is sent from the device to the host, while OUT is for sending data to the device. A transaction is a single communication that cannot be interrupted by any other communication on the bus [9].

Every transaction has at least one phase or a packet. All of the packets have a unique PID. The token packet can contain either a Setup, OUT, IN or Start-of-Frame (SOF). The data phase contains data toggle bits and data, and the handshake packet contains various handshake status bits such as ACK, NAK, STALL.

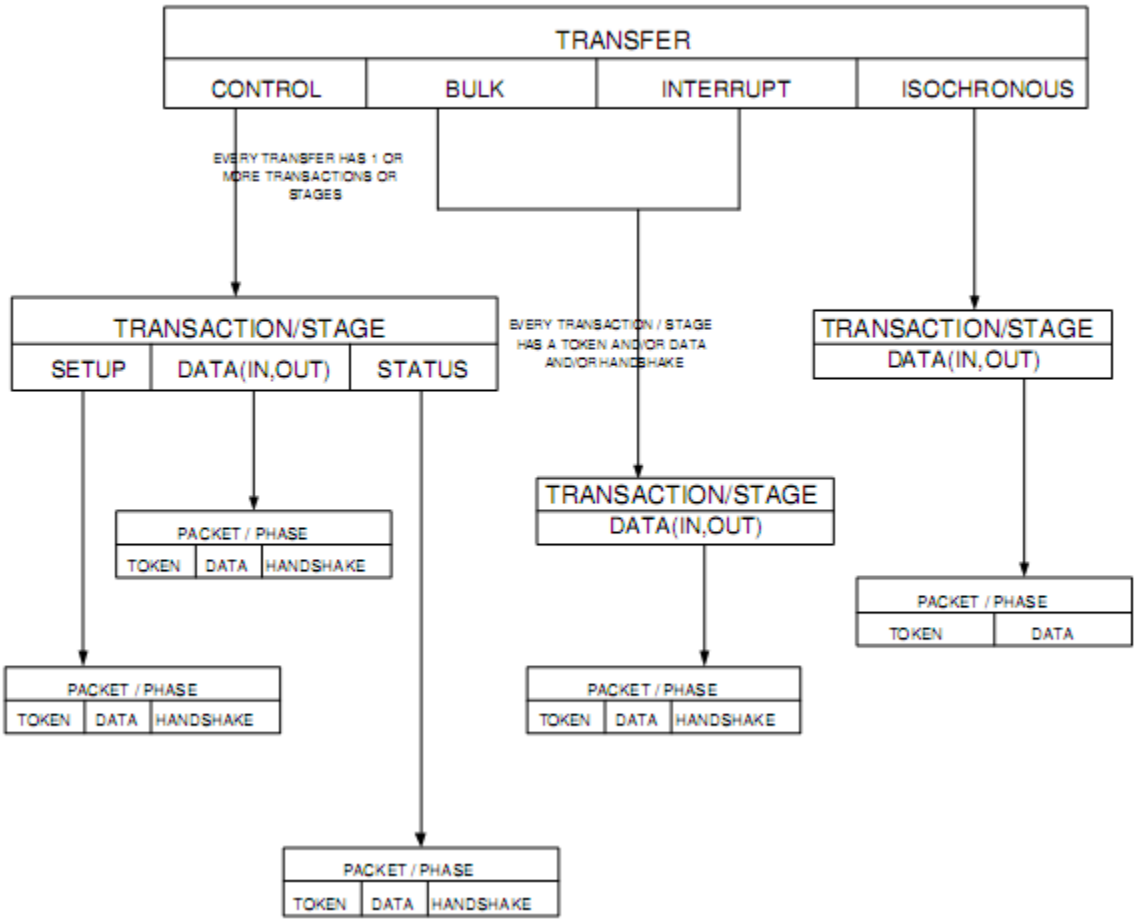

Figure 3.4: Various components of USB communications

3.5 Types of Transfers

USB provides users with the freedom of designing devices of varying speed and characteristics. In order to design more efficient and reliable USB devices, it is very important to understand the different transfer methods supported by the USB. The four transfer types supported by USB 1.1 are

(1) Control Transfers

(2) Interrupt Transfers

(3) Bulk Transfers

(4) Isochronous Transfers

3.5.1 Control Transfers

The primary purpose of a control transfer is to learn more about the device's configuration. Every device must support the control transfers over the default message pipe. The default message pipe has an endpoint number 0. Control transfers always use message pipes.

Data Format in Control Transfers: It is important to understand the data format used in control transfers since they use the message pipes. As was shown in our earlier discussion, a message pipe requires data to be in a

USB-defined structure. All control transfers have a setup stage, status stage and an optional data stage. Figure 3.5 shows a setup transaction for a control write transfer.

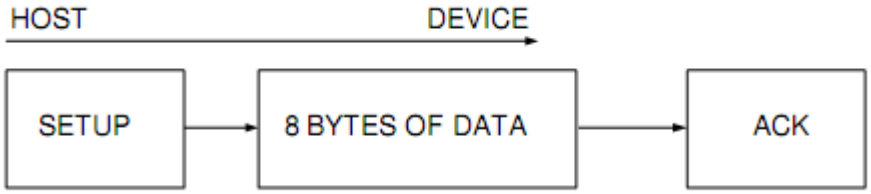

Figure 3.5: Setup Transaction using Control Transfers

During the setup transaction, the host initially sends a packet that has the PID, which identifies whether it is a control transfer or not. This PID also contains the direction of the data. The host then sends a data packet 8 bytes in length and waits for the device to send an acknowledge (ACK) packet. There are certain constraints on the data packet size. The maximum size for a low-speed device is 8 bytes. For full and high speed-devices data packets can be 8, 16, 32, or 64 bytes in length.

In the case of control transfers, the host controller reserves bandwidth for the transfers, usually it is 10 percent for low-speed devices and 20 percent for high speed devices.

Error Handling: In control transfers, an error condition is signaled if an ACK packet is not received. During the error condition, the host tries to send the data again. This process is repeated a maximum of three times, and if it still does not receive any response from the device, then the host STALLs the communication until the problem is corrected.

3.5.2 Interrupt Transfers

Interrupt transfers are generally used for transferring small chunks of data at random times. They are often used in keyboards. Interrupt transfers are capable of retransferring the data in event of any errors in the USB bus. Interrupt transfers use stream pipes, which allow the data to flow only in one direction. The maximum data packet that can be transferred over an interrupt pipe is 64 bytes. In interrupt transfers, endpoints must specify the required bus access period. This period can ranges from 1 ms to 255 ms. Data is transmitted in an alternating data toggle bit pattern. The maximum data packet in a high speed device using interrupt transfer is 1024 bytes [9].

3.5.3 Bulk Transfers

A bulk transfer is used to send large amounts of data, where timing is not a constraint. Most printers are designed using this transfer method. The bulk transfers have one or more IN or OUT transactions. The maximum data that can be transferred is 512 bytes in a high speed device. If the entire bandwidth is occupied by interrupt, control, and isochronous transfers (explained in Section 3.5.4), then bulk transfers need to wait until bandwidth becomes available.

3.5.4 Isochronous Transfers

The CID II project employs this method of transfers. Isochronous data transfers are an ideal method of transfers for time-relevant data. The CID II is a real-time device, which requires guaranteed bandwidth on the USB bus [4]. Isochronous transfers do not provide error checking and assumes that errors do not occur. An isochronous transfer is capable of sending 1023 bytes of data per millisecond frame. The isochronous transfers have the least overhead when compared to other non-isochronous transfers [4]. Only full speed and high speed devices support isochronous method of transfers.

Isochronous transfers are guaranteed with bandwidth; hence the host controller manages the bandwidth allocation for the isochronous devices. In USB 1.1, a full speed device with 1023 bytes per frame occupies 69 percent of the USB's bandwidth [9]. If another isochronous device requires the same amount of bandwidth then the host controller does not allocate the bandwidth until the transfer for the first device is completed. However if the requested bandwidth is less than the available bandwidth, then the device is allocated with the resources. Isochronous pipes are stream pipes and hence they are uni-directional. The CID II project has 0 and 1 for its output and input isochronous pipes respectively. Isochronous pipes usually move data every 1 ms packet frame, this rate guarantees the speed for streaming audio and video applications.

Chapter 4 Designing the Controller Interface Device Driver

The USB driver code for the CID II consists of several different subroutines. The DDK also provides functions that can be used in any Windows based device driver. The important routines written for the CID II device driver are described in this chapter. One of the goals of this project is to reduce the latency of 6 ms delay involved with the general purpose driver. The important steps that we have employed in order to reduce the latency in the device driver are

(1) Designing a custom driver that performs only isochronous transfers: This strategy eliminates the unnecessary overhead that would have been introduced by a general purpose driver.

(2) Performing isochronous transfers in an efficient way. Section 4.6 describes more about the isochronous transfer techniques employed in this device driver.

4.1 Control Flow in a Device Driver

It is a somewhat difficult to describe the exact flow of control in a device driver, since the driver has to perform various functions as a driver and also coordinate with the operating system. The diagram in Figure 4.1 shows the interaction of different modules in the device driver for the CID II. The figure also shows the file names for the modules. There are direct and indirect function calls made to the different modules in Figure 4.1.

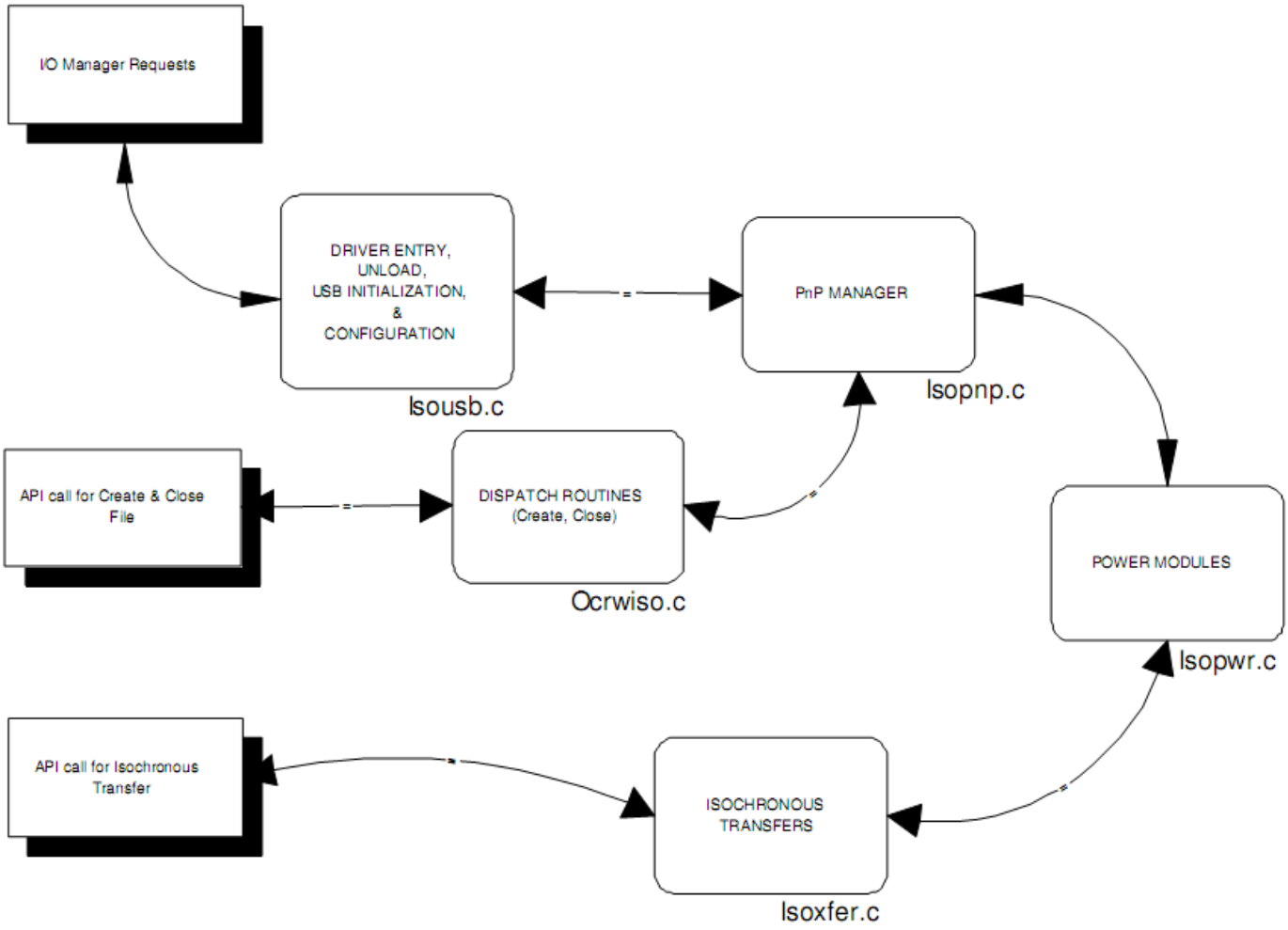

Figure 4.1: Interaction of different modules in the CID II device driver

4.2 Driver Entry Routine

The driver entry routine is the main entry point for any kernel mode device driver. This routine is responsible for initializing the Driver Object. The I/O Manager calls this routine at PASSIVE_LEVEL of IRQL (Interrupt Request Level) when it loads the driver.
The syntax of driver entry routine is as shown in the code segment below:

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
                     IN PUNICODE_STRING RegistryPath ){


   DriverObject->DriverExtension->AddDevice = PnPAddDevice;
   DriverObject->DriverUnload = Unload;
.............
}
```

This function supplies the entry points for most of the driver's standard routines. The **PnPAddDevice** is the driver's *AddDevice()* routine. This is also where the driver's *Unload()* routine is initialized.

The return value **NTSTATUS** indicates whether or not the driver was successfully loaded and if it is available to process requests.

The **NTSTATUS** can return success values, informational values, warnings, and error values. All of the different **NTSTATUS** return values are defined in ntdef.h. The following system supplied macros can be used to check the return value.

(1) **NT_SUCCESS(Status):** This macro evaluates to TRUE if the return value specified by Status is success.

(2) **NT_INFORMATION(Status):** This macro evaluates to TRUE if the return value specified by Status is an informational type.

(3) **NT_WARNING(Status):** In event of a warning this macro returns TRUE.

(4) **NT_ERROR(Status):** This macro evaluates to TRUE if the return value specified by Status is an error type.

Driver writers can also define and provide their own **NTSTATUS** values, but that is beyond the scope of this research.

This chapter tries to explain in detail all the important routines that are invoked from the Driver entry routine.

4.3 Working with I/O Request Packet (IRP)

The IRP is a data structure used by the I/O Manager to process an I/O request. The IRP structure consists of a several fields, which include memory descriptor list, flags, user buffers, address of cancelled routines and current stack locations. When an IRP is created by the I/O Manager, an array of stack locations is also created. The stack location serves as an information bank for the IRP. The basic I/O stack is shown in the Figure 4.2. An IRP starts its journey at the time it is created by the I/O Manager and travels through various driver routines. IRPs do not necessarily need to be processed by all the routines, sometimes they are just passed down to the next level driver. The code that passes the IRP is listed below:

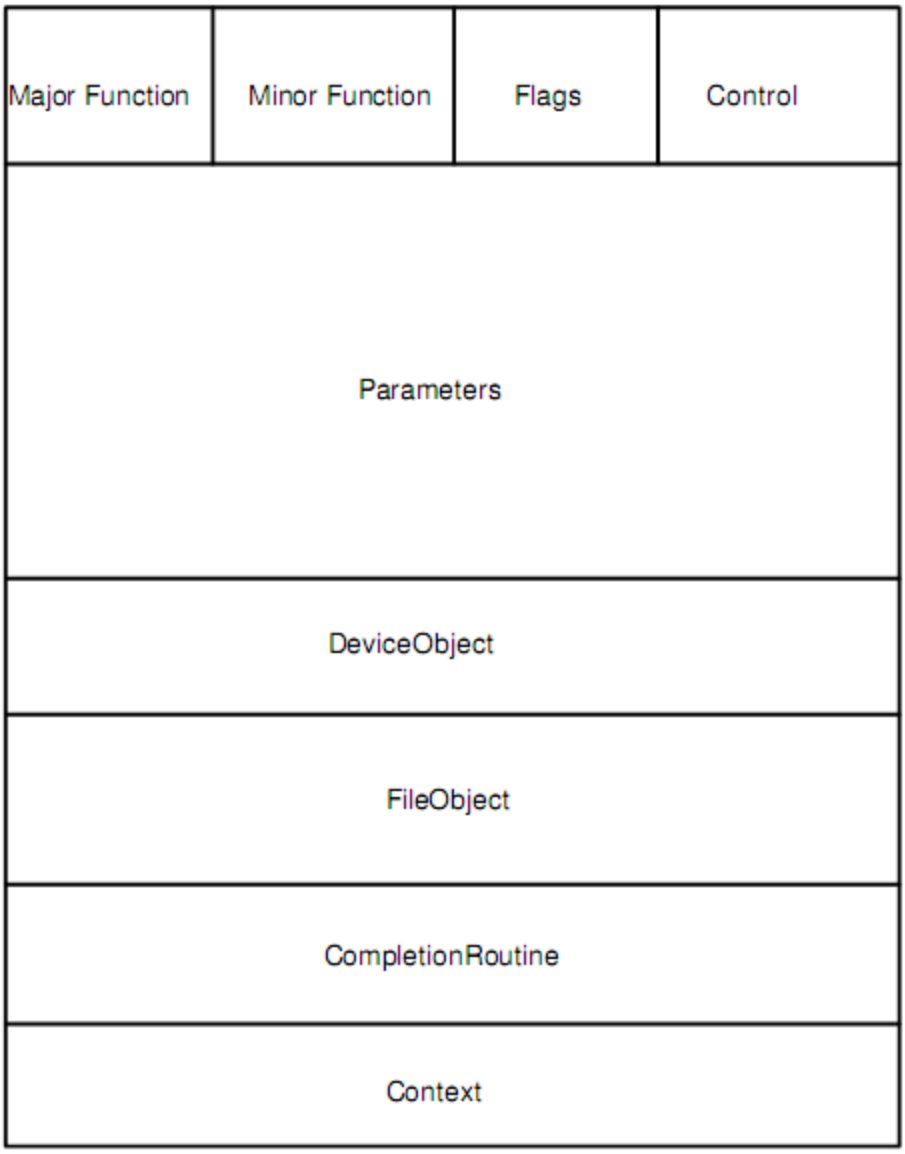

Figure 4.2: I/O Stack Location for the IRP

```
PDEVICE_OBJECT DeviceObject;
PIO_STACK_LOCATION stack =IoGetNextIrpStackLocation(Irp);
stack->MajorFunction = IRP_MJ_Xxx;
NTSTATUS status = IoCallDriver(DeviceObject, Irp);
```

Once an IRP is created, a call to the function *IoGetNextIrpStackLocation(Irp)* obtains a handle to the first location of the stack. Every IRP has a major and a minor function code, the major code is assigned to the stack location. A call to the function *IoCallDriver()* passes the IRP to the driver.

Now that the IRP is passed down to the next level driver, it is important to know the result of that operation. A completion routine is installed to obtain information about the IRP that is passed down. The function

IoSetCompletionRoutine implements the completion routine required for an IRP. Occasionally the I/O Manager may need to cancel an IRP. A call to the function *IoCancelIrp* terminates the IRP.

4.4 Different I/O Request Packets (IRPs) and their Major Codes

The driver entry routine lists the function pointers for different Win32 calls. All the Win32 function calls have specific IRP major code, such as: IRP_MJ_DEVICE_CONTROL for a *DeviceIoControl* call and IRP_MJ_CREATE represents the IRP major code for a *CreateFile* function. Table 4.1 shows the complete IRP major codes for all the function calls of the driver entry routine used in the CID driver.

Table 4.1: The IRP Major Codes and Their Respective Win32 Function Calls

IRP Major Code	Win32 Function Call
IRP_MJ_CREATE	CreateFile
IRP_MJ_CLOSE	Close File
IRP_MJ_DEVICE_CONTROL	DeviceIoControl
IRP_MJ_PNP	Plug and Play
IRP_MJ_POWER	Power Management

The Plug and Play IRP also has some minor codes that any PnP driver supports.

4.5 The Plug and Play IRPs

This section discusses how the driver handles various plug and play calls. The PnP implementation for the CID driver is listed in the Isopnp.c file. The PnP manager is responsible to generate the correct minor function codes for driver writers. The major function code for the PnP IRP is IRP_MJ_PNP. It is important to verify the major code before handling the minor PnP calls. The macro *ASSERT* is used to evaluate various error conditions. The most common PnP IRP minor function codes are listed below.

- IRP_MN_START_DEVICE*
- IRP_MN_QUERY_REMOVE_DEVICE
- IRP_MN_REMOVE_DEVICE*
- IRP_MN_CANCEL_REMOVE_DEVICE
- IRP MN STOP DEVICE*
- IRP_MN_QUERY_STOP_DEVICE
- IRP_MN_CANCEL_STOP_DEVICE
- IRP_MN_SURPRISE_REMOVAL
- IRP_MN_QUERY_CAPABILITIES*

*The IRPs that are implemented in the CID driver.

When a device is attached to the system, the PnP manager signals the lowest level driver or the bus driver to create a physical device object (PDO). A functional device object (FDO) is also created at the function driver level and a filter device object (fDO) is created by the filter driver if any filter drivers exist. All of the above objects are created and initialized when a call to driver's AddDevice() routine is made. If the call to the *AddDevice()* routine fails then the next lowest driver issues a *RemoveDevice* message.

The main function that handles the PnP calls is *DispatchPnp().* The pseudo code for the function is illustrated below:

```
NTSTATUS DispatchPnp( pointer to the device object, pointer toIRP)
 {
   PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
   if (!LockDevice(DeviceObject))
      return CompleteRequest(Irp, STATUS_DELETE_PENDING, 0);
   irpStack = IoGetCurrentIrpStackLocation (Irp);
   fcn = irpStack->MinorFunction;
   switch (fcn)
 {
     case IRP_MN_START_DEVICE:
         ntStatus = HandleStartDevice(DeviceObject,Irp);
// if error conditions, handle appropriately
         break;
       case IRP_MN_STOP_DEVICE:
         ntStatus = StopDevice(DeviceObject);
// if error conditions, handle appropriately
         break;
       default:
          ntStatus = DefaultPnpHandler(DeviceObject,Irp);
          break;
       ...........
}
```

This code sample obtains a handle to the IRP stack which contains all of the minor function codes. A switch statement is used to jump to different minor functions. A sub dispatch function is then written for every minor function listed in the dispatch routine. Any unrecognized minor functions are handled by the *DefaultPnpHandler()* routine.

All of the data from the PnP manager is passed to the driver in the form of an IRP. If an IRP is received that is not needed for a particular driver then this IRP is simply passed down the device stack for processing by other lower device drivers. The *DefaultPnpHandler()* function is a similar routine which passes down the IRP. The pseudo code for the DefaultPnpHandler() function is illustrated below:

```
NTSTATUS DefaultPnpHandler(IN pointer to the device object, IN point to the Irp )
{
    pdeviceExtension = DeviceObject->DeviceExtension;
    IoSkipCurrentIrpStackLocation(Irp);
    return IoCallDriver(pdeviceExtension->StackDeviceObject, Irp);
```

```
}
```

Here the call to *IoSkipCurrentIrpStackLocation()* will modify the I/O Stack location, so that if the current driver makes a call to the next lower driver, it obtains the pointer to the same IRP the current driver obtained. This allows the IRP to be passed down the stack. The *IoCallDriver()* function returns the appropriate *NTSTATUS* return value after passing the IRP down the stack.

4.5.1 Plug and Play Minor Functions

IRP_MN_START_DEVICE: This helper function starts the device. However before the device is started, the PnP manager needs to perform some tasks. When a hardware device is initially attached, the PnP manager searches the registry to find out what kind of drivers need to be loaded for the hardware device to function. The PnP manager obtains a list of resources required for the device and sends the **IRP_MN_START_DEVICE** as a IRP. The function drivers accept this IRP and configures the device.

The function *DefaultPnpHandler()* provides information about an IRP being passed down without much worry about the result. When starting a device it is important to know the result of passing down an IRP and this is illustrated in the SendAndWaitUrb() routine. The prototype of this routine is shown below:

```
NTSTATUS SendAndWaitUrb(IN PDEVICE_OBJECT DeviceObject,IN PIRP Irp)
{
KEVENT event; PDEVICE_EXTENSION pdeviceExtension;
pdeviceExtension=(PDEVICE_EXTENSION)DeviceObject->DeviceExtension;
NTSTATUS ntStatus;
// Check if we are at the PASSIVE LEVEL
   ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
   KeInitializeEvent(&event, NotificationEvent, FALSE);
// make a copy of stack for next driver
// since we are going to install a completion routine
IoCopyCurrentIrpStackLocationToNext(Irp);
IoSetCompletionRoutine(Irp,
                       (PIO_COMPLETION_ROUTINE)OnRequestComplete,
                       (PVOID) &event,
                       TRUE,TRUE,TRUE);
//call the lower driver
ntStatus = IoCallDriver(pdeviceExtension->StackDeviceObject, Irp);

  if (ntStatus == STATUS_PENDING)
     {
  KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
  ntStatus = Irp->IoStatus.Status;
     }
  return ntStatus;
}// end of SendAndWaitUrb()
```

This function is written to regain control of IRP_MN_START_DEVICE request. Here the dispatch routine waits for a kernel event to indicate the completion of an IRP in the lower layers. The *SendAndWaitUrb()*

function creates a kernel event and then the function call to *KeInitializeEvent()* initializes the event object. It is important that the kernel event be initialized at the PASSIVE_LEVEL_IRQL. The objective is to know the completion of the IRP, so a completion routine is installed. The *IoCallDriver()* passes the IRP to the next lower driver and the *KeWaitForSingleObject()* function waits on the kernel event for the IRP completion.

The *FowardAndWait()* routine is called from a sub-dispatch routine which is specifically written for each device type. A prototype of the sub-dispatch function is shown below:

```
NTSTATUS HandleStartDevice(DeviceObject,Irp) {
      NTSTATUS ntStatus;
//Pass it down until it finds out which driver needs to handle

      ntStatus = SendAndWaitUrb(DeviceObject, Irp);
      ntStatus = Start_CID_Device(DeviceObject);
      return CompleteRequest(Irp, ntStatus, 0);
}
```

The *HandleStartDevice()* function includes a new Start_CID_Device() function which configures the device. The implementation of the *StartDevice()* routine is device dependent. The implementation of this routine in the CID II device driver is a similar to that for any other USB based driver.

The functional USB device driver written for the CID II never talks directly to the device. The functional driver creates a request block known as USB Request Block (URB) and submits it to the USB bus driver (USBD). An URB is created by allocating memory from the heap using the *ExAllocate()* function. The following code helps to understand the initialization process for a USB device.

```
NTSTATUS Start_CID_Device(DeviceObject)
 {
      PUSB_DEVICE_DESCRIPTOR dd = NULL;
// create a pointer to the USB Request Block
      PURB urb;
      ULONG size;
urb = ExAllocatePool( NonPagedPool,
          sizeof(struct _URB_CONTROL_DESCRIPTOR_REQUEST));
UsbBuildGetDescriptorRequest(urb,
  (USHORT) sizeof (struct _URB_CONTROL_DESCRIPTOR_REQUEST),
   USB_DEVICE_DESCRIPTOR_TYPE,
     0,
     0,
     dd,
     NULL,
     size,
     NULL);
//now submit the URB to USBD.sys as an internal IOCTL
     ntStatus = Isousb_CallUSBD(DeviceObject, urb);
```

```
// if success in reading the device, configure the device
if (NT_SUCCESS(ntStatus)) {
     ntStatus = CID_ConfigureDevice(DeviceObject);
   }
}// end of Start_CID_Device
```

This code is used to extract the USB device descriptor values, which include the pipe type, serial number, and vendor details.

- The function *UsbBuildGetDescriptorRequest()* is a macro that generates inline statements to initialize the fields of the get descriptor request substructure and is defined in USBDLIB header file.
- In the *UsbBuildGetDescriptorRequest()* function, the first two parameters are pointers to the URB and the size. The third parameter indicates the descriptor type. The next two values are zero since they represent the index and language ID fields. The NULL values indicate that we are not passing any transfer buffers.
- The next task of the routine is to submit this URB to the bus driver as an internal I/O control (IOCTL). This is done by the function *Isousb_CallUSBD().*

```
NTSTATUS Isousb_CallUSBD(DeviceObject,Urb) {
   NTSTATUS ntStatus, status = STATUS_SUCCESS;
KeInitializeEvent(&event, NotificationEvent, FALSE);
   IO_STATUS_BLOCK ioStatus;
   PIO_STACK_LOCATION nextStack;
//prepare the irp for submitting to the bus driver
irp = IoBuildDeviceIoControlRequest(
          IOCTL_INTERNAL_USB_SUBMIT_URB,
          pdeviceExtension->StackDeviceObject,
          NULL,
          0,
          NULL,
          0,
          TRUE, /* INTERNAL */
          &event,
          &ioStatus);
nextStack = IoGetNextIrpStackLocation(irp);
// this call submits the irp to the Bus Driver
ntStatus = IoCallDriver(pdeviceExtension->StackDeviceObject,irp);
// Wait if the status is pending
if (ntStatus == STATUS_PENDING){
status=KeWaitForSingleObject(&event,Suspended,KernelMode,FALSE,
                  NULL);}
    return ntStatus;
}
```

This routine creates an IRP and submits it as an internal IOCTL to the USB bus driver (USBD). It waits until the IRP has been processed. This routine is used whenever there is a need to submit an IRP to the USBD. An internal IOCTL IRP is built using the function *IoBuildDeviceIoControlRequest().* A call to the *IoCallDriver()* function is made, which essentially submits the IRP to the USBD. Once the USBD is done processing the

request, the I/O manager will signal our event and delete the IRP. However, the driver needs to wait until this is done.

IRP_MN_STOPDEVICE: Once a device is started, the I/O manager may request the device to stop. In Windows 2000, the user can request to stop the device before unplugging the device physically. The I/O manager requests a *STOP_DEVICE* in order to reassign the resources. A sub-dispatch function is written to handle the request for the stop device IRP. This handler function deals with cancelling any queued IRPs and logically the handler function does exactly the opposite of configuring the USB device.

4.6 Isochronous Transfers

The CID II requires transfers to be done every millisecond. The isochronous method of transfer is the answer for this type of communications. Isochronous transfers are able to provide a guaranteed latency by reserving 90 percent of the USB bandwidth on the USB system. Hence care must be taken to use this bandwidth. Isochronous transfers can be used to transfer data in discrete mode, where the data is sent and received without any queuing or buffering. This technique may not be suitable for the CID II device driver. Hence we employ a technique where we buffer the requests and submit all the requests at once to the bus driver. This technique ensures that the data is being sent or received every 1 millisecond. We created several data structures to support this operation. To perform the read and write operations, we created a data structure that consists of the following fields.

```
typedef struct _RWISO_BUF_OBJECT
{
   PDEVICE_OBJECT DeviceObject;
   ULONG packetsize;
   ULONG number_of_packets;
   PUSBD_PIPE_INFORMATION readwrite_pipeinfo;
   PVOID trans_buf;
   ULONG trans_bufl;
   PVOID IsoDescriptorBuffer;
ULONG frame_buf;
   ULONG nbuf;
   PISO_TRANSFER_OBJECT TransferObject;
};
```

After all the fields are initialized, the read/write object is submitted to the bus driver. The initialization routines are described in the following sections.

4.6.1 Bandwidth Allocation

In the *Start CID Device()* routine we has the default configuration of the USB device, but this default setting does not identify the type of pipes used for the transfer. Hence no actual bandwidth is reserved in the *Start_CID_Device* routine. To solve this problem, a simple helper routine *Set_Interface()* was written. This function reserves the bandwidth for the isochronous endpoints.

```
NTSTATUS SetInterface(DeviceObject,InterfaceNumber,AlternateSetting)
{ interfaceDescriptor = USBD_ParseConfigurationDescriptorEx(
                            configurationDescriptor,
                            configurationDescriptor,
                            InterfaceNumber,
                            AlternateSetting,
                            -1,-1,-1);
urbSize = GET_SELECT_INTERFACE_REQUEST_SIZE(numberOfPipes);

UsbBuildSelectInterfaceRequest(urb,
                                urbSize,
                                pdeviceExtension->ConfigurationHandle,
                                InterfaceNumber,
                                AlternateSetting);
   for (i = 0 ;i < numberOfPipes ;i++ )
  {
    interfaceInformation->Pipes[i].MaximumTransferSize = (64*1024) -1;
  }
// Call to the Bus driver
ntStatus = Isousb_CallUSBD(DeviceObject, urb);
    return(ntStatus);
}
```

The function call *USBD_ParseConfigurationDescriptorEx(),* with interface 0 and alternate setting 1 returns a count of the number of endpoints used. The function call *GET_SELECT_INTERFACE_REQUEST_SIZE(),* calculates the number of bytes required to hold a select interface request. This function is also used to open a specified number of pipes, and a parameter is passed to the function call to perform this job. For purposes of simplicity, the code that allocates memory is not shown in the above illustration. The function call *UsbBuildSelectInterfaceRequest()* is used to build a URB to select alternate setting 1 of the interface number 0. It is important to set the maximum transfer size of each pipe to prevent errors in the driver.

4.6.2 Initialization Routine for the Isochronous Transfer

All isochronous transfers are broken down into packets. The size of the packet is dependent upon the maximum size of the endpoint. The following code fragment shows only the important functions used to create and initialize a USB Request Block (URB) for the isochronous transfer. A complete listing is provided in Appendix A.

```
// allocate and prepare the URB
urbSize = GET_ISO_URB_SIZE(number_of_packets);

urb->UrbHeader.Length = urbSize;
urb->UrbHeader.Function = URB_FUNCTION_ISOCH_TRANSFER;
urb->UrbIsochronousTransfer.PipeHandle = pipeInfo->PipeHandle;
//determine the direction of transfers
if(read)
   urb->UrbIsochronousTransfer.TransferFlags =
    USBD_TRANSFER_DIRECTION_IN;
else
   urb->UrbIsochronousTransfer.TransferFlags=
    USBD_TRANSFER_DIRECTION_OUT;
   urb->UrbIsochronousTransfer.TransferFlags |=
    USBD_START_ISO_TRANSFER_ASAP;
   urb->UrbIsochronousTransfer.TransferFlags |=
    USBD_SHORT_TRANSFER_OK;
// setup the ISO packet descriptors
   for (i=0; i<number_of_packets; i++)
  {
    urb->UrbIsochronousTransfer.IsoPacket[i].Offset =
                      i * streamObject->PacketSize;
    urb->UrbIsochronousTransfer.IsoPacket[i].Length =
                      streamObject->PacketSize;
}
// allocate and prepare the IRP
irp = IoAllocateIrp(stackSize, FALSE);
IoInitializeIrp(irp, irp->Size, stackSize);

nextStack->Parameters.DeviceIoControl.IoControlCode =
   IOCTL_INTERNAL_USB_SUBMIT_URB;
nextStack->MajorFunction = IRP_MJ_INTERNAL_DEVICE_CONTROL;

IoSetCompletionRoutine(irp,
              IsoTransferComplete,
              transferObject,
              TRUE,
              TRUE,
              TRUE);
KeInitializeEvent(&transferObject->Done, NotificationEvent, FALSE);
   return STATUS_SUCCESS;
}
```

All of the packet descriptors are read or written to the data buffers, which are contiguous in the memory. This type of contiguous memory requires information about offset and length. The host controller driver is responsible for setting the length of the packet descriptor and the *InitIsoObject()* function sets the offset. The contents of the isochronous packet in the USB bus are stored in the *USBD_ISO_PACKET_DESCRIPTOR* structure. Isochronous transfers employ certain built-in functions that aid the transfers; these include the *URB_FUNCTION_ISOCH_TRANSFER* function code which identifies if the transfer is an isochronous transfer.

IsoPacket is an array of *USBD_ISO_PACKET_DESCRIPTOR* structures. The function code *GET_ISO_URB_SIZE* gives information regarding the number of bytes required to hold for an isochronous transfer. The *START_ISO_TRANSFER_ASAP* setting for the flag requests the transfers to start as soon as possible, and is also an alternate way to specify the 32-bit ULONG *StartFrame* number.

After all the objects are initialized, a call to the USB Bus Driver (USBD.sys) performs the actual transfer. All of the traffic in the USB can be monitored using a wide range of devices available. These devices provide wealth of information regarding the data on the bus and they can also be used to debug errors.

4.7 Building, Testing and Installing Device Drivers

Once the driver is written, it must be compiled. The process of compiling a driver is called building the driver. Chapter 5 and 6 illustrate the concepts of building, testing, and installing the driver for the CID II.

This chapter has listed some of the important routines of the CID driver. The complete driver includes many other routines, so the driver has many lines of code. Hence a copy of the driver can be obtained from the author or from Dr. Brian K. Johnson (b.k.johnson@ieee.org).

4.8 Summary

This chapter is an attempt to explain most of the important code segments of the CID II device driver we have written. This chapter also described the technique we have implemented in the driver to perform the isochronous method of transfers. The general purpose drivers are written to ensure the functionality of the device, hence they may not concentrate on efficiency of a particular transfer. A custom designed driver can eliminate a good amount of overhead associated with any general purpose device driver.

Chapter 5 Techniques to Build, Test and Debug a Windows Device Driver

A device driver is not always written from scratch. Users may be required to do some minor modifications to the driver source code to get the device working. Even small changes to the driver source code require recompilation of the driver. In order to perform such tasks, the user need not necessarily have a great understanding of device driver programming. This chapter provides information for both driver writers and non-driver writers to compile and debug their drivers.

5.1 Building Windows Device Drivers

Even though drivers are similar to any other software code, there are no Integrated Development Environments (IDEs) available to compile and debug drivers. The driver compilation is done using the *Build* utility provided with the Driver Development Kit (DDK).

The DDK [11] provides two methods to build a driver: the *checked* version and the *free* version. The free build is used for a retail and optimized release of a driver, when all testing is done. However, checked build is used mostly for debugging and testing the driver. The checked build has different error checking options and

verification procedures that aid efficient driver development. On the downside, checked build consumes more memory and hard disk space.

5.1.1 Build Utility

In order for the build utility to work, the following must be installed in the order they are mentioned

(1) Microsoft Visual C++: Required for the libraries

(2) Windows DDK: Installs checked and free build utilities

After the installation is complete, checked build is invoked from the Start Menu in Windows. This gives the user with a command prompt a window with DDK installation directory as its current directory. The build utility is invoked by typing the build command with the following option.

build -cZ

Here the command line option -c specifies to do a clean build by deleting the .obj files. The -Z option inhibits dependency checking of source and header files. These command line options are sufficient enough to build a driver, but advanced users can make use of other build utility options. All the syntax errors, typos, and various errors if any, are displayed after the build command.

5.2 Debugging device drivers

Debugging drivers has various advantages:

- Debugging helps to identify the problem when the drivers are not functioning correctly.
- Debugging drivers is a way to learn the internals of drivers.
- Hardware devices are more reliable if they have efficient and correct drivers.

Thorough testing of a driver is important before releasing the driver. All the functions must be tested to see if they return correct values. Various checks can be made for boundary conditions, especially when dealing with packet sizes in USB communications. Drivers should be installed on a computer with a slower CPU and their performance should be monitored.

5.3 Different Types of Errors

Improper driver writing techniques can result in a system crash, or even worse, can damage the hardware device. A system crash due to a device driver results in a blue screen with some information displayed. The information provided during a crash is not really very useful unless advanced debugging techniques are used. A core dump can be made in event of a crash. This core dump has a lot of information regarding the crash. Memory leaks are another common source of errors. Leaks occur when memory allocated is not freed after use. In order to understand the causes for a driver to crash, it is important to understand and know how to use the debugging techniques available.

5.4 Debugging Routines and Techniques

Several debugging techniques were used in debugging the CID II driver. One simple way was to print trace statements; these statements indicate the return values of various functions in the driver. Various conditional flags were set which could be used to determine errors. There are several kernel-mode debugging routines available in the DDK. These functions can be used either in the free or checked build versions of the driver. Two sets of functions were mainly used, the functions that begin with DbgXxx can be used in both checked as well as the free versions. The functions that begin with KdXxx can be used only in checked build of a driver. The macro that was commonly used for this project is *DbgPrint()* which facilitates both the checked and the free versions of the driver.

There is also an *ASSERT* macro, which tests an expression, and if the expression is false, it enables the debugger. This macro is defined in the ntddk.h and wdm.h header files.

5.5 Debuggers for Windows Operating System

The standard debugger that comes with the Microsoft DDK is the *Windbg* debugger. Microsoft also provides a console program for debugging kernel-mode drivers called *KD*. There are also other commercial debuggers that can do the job. The SoftIce debugger from Numega is a very good debugger that can do kernel debugging on a stand-alone system. Again the choice of a debugger is left to the convenience and financial resources of the user. This project employed *Windbg* for all debugging.

5.5.1 Installing and Configuring the Debugger

The DDK provides an option to install the debugging tools during the DDK installation. Also, more recent versions of the debugger can be downloaded from Microsoft's website.

The hardware setup for kernel mode debugging is shown in Figure 5.1. The use of Windbg requires two machines, where the target machine has the CID II connected to it. The device driver is installed on the target machine. The host computer is running the *Windbg* debugger and also has the required symbol files installed. The two machines are connected to each other using a null modem cable. A null-modem cable is a serial cable capable of sending data between the serial ports of the computers.

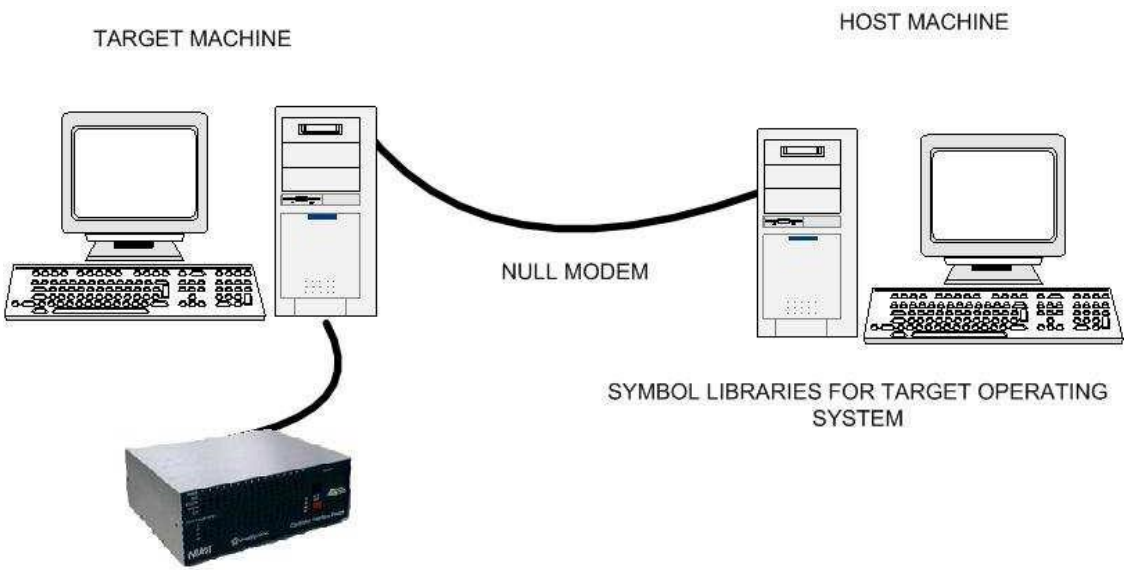

Figure 5.1: Setting up kernel debugging.

The connectivity between the computers can be verified using the Hyperterminal program provided with the Windows operating system. The details of the exact configuration and the batch files used are mentioned in Appendix A.

In order to debug any target machine, the host system must have the symbol files matching the target operating system. These symbol libraries have a wealth of information about various global and local variables used in any software code. In order to save disk space, Microsoft does not ship these symbol libraries with the Windows operating system. Instead they provide only system binaries, which are not sufficient to debug the drivers. However all the symbol files can be installed from the Customer Support Diagnostics CDROM. Once the symbols are installed on the host machine, it is important that the debugger finds the path where the files are installed.

The symbol path can be set using the MS-DOS command set. Since multiple set commands are used, a batch file can be created. The batch file is listed below:

```
set _NT_SYMBOL_PATH=c:\winnt\symbols
set _NT_DEBUG_PORT=com1
set _NT_DEBUG_BAUD_RATE=19200
set _NT_LOG_FILE_OPEN=c:\DEBUG.LOG
```

In this case, the symbols are installed to the default directory, but if the user decides to install to a different directory then that change must be appropriately reflected in the batch file. The serial port used was COM1, and the baud rate was set at 19200 kbps, which worked very well for debugging the driver.

After the path was set successfully, *Windbg* was started and kernel debugging was selected. The target machine was restarted and upon booting various messages are displayed on the Windbg window. There are various other commands that are available in *Windbg* which are listed in the help menu of *Windbg* program.

Chapter 6 Installing the CID II Windows Driver Model (WDM) Driver

6.1 Installing a Windows Device Driver

Many new features are available in new operating systems. All of the latest Microsoft Windows Operating systems include drivers for the most common hardware devices, which essentially makes the hardware PnP. A device information file, commonly known as an INF file is used to install a driver that is not provided with the operating system. The INF file has all of the information required to install the hardware device for a Windows operating system. The INF files have .INF file extensions.

During the installation, the INF file is copied into the windows\inf directory. INF files have various sections and each section is written on a new line with the section name enclosed in square brackets, and comments if any, are written starting with a semicolon.

6.2 Sections of a INF file

The common sections of any INF file contain version information, manufacturer name, and the destination directory to which the driver file is copied. The following is an excerpt from the CID.INF file

```
; Installation inf for the CID [Version] Signature="$CHICAGO$"
Class=USB
provider=%McCain-NIATT%

[SourceDisksFiles] CIDII.sys = 1 CIDII.inf = 1

[SourceDisksNames] 1="Controller Interface Device Installation
Disk",,,

[DestinationDirs] CID.Files.Ext = 10,System32\Drivers
CID.Files.Inf = 10,INF

[Manufacturer]
%McCain-NIATT%=McCain-NIATT

[McCain-NIATT]
%USB\VID_c590&PID_0250.DeviceDesc%=CID.Dev, USB\VID_c590&PID_0250
```

This excerpt shows information regarding the version and manufacturer.

The *Signature* in the [VERSION] section has a magic value $Chicago$, $Windows NT$, or $Windows 95$. The device class is listed in the next line, which is USB in the case of CID II. The [*SourceDisksFiles*] section has the names of both the driver (CIDII.sys) and the INF file names; these are each assigned with a value 1, which indicates the system to search for files on disk number 1. The [SourceDisksNames] section provides the name of the disk on which the files are found. The target directory to which the windows driver installed is \System32\Drivers. The numeric value 10 helps to identify the correct Windows directory if the user has installed Windows 2000 to a non-default directory. The value *10,System32\Drivers* identifies the driver's

directory in both Windows 2000 and Windows 98. The manufacturer section lists the hardware device's vendor name.

There are various other sections in an INF file that make the driver more compatible and easy to install. In the following, a listing describes additional sections of an INF file.

```
[CID.Dev] CopyFiles=CID.Files.Ext, CID.Files.Inf AddReg=CID.AddReg

[CID.Dev.NT] CopyFiles=CID.Files.Ext, CID.Files.Inf
AddReg=CID.AddReg

[CID.Dev.NT.Services] Addservice = CID, 0x00000002, CID.AddService

[CID.AddService]
DisplayName      = %CID.SvcDesc%
ServiceType      = 1        ; SERVICE_KERNEL_DRIVER
StartType        = 2        ; SERVICE_AUTO_START
ErrorControl     = 1        ; SERVICE_ERROR_NORMAL
ServiceBinary    = %10%\System32\Drivers\CIDII.sys
LoadOrderGroup   = Base

[CID.AddReg] HKR,,DevLoader,,*ntkern HKR,,NTMPDriver,,CIDII.sys

[CID.Files.Ext] CIDII.sys

[CID.Files.Inf] CIDII.Inf

;-------------------------------------------------------------;
[Strings] McCain-NIATT="McCain-NIATT CID II"
USB\VID_c590&PID_0250.DeviceDesc="Controller Interface Device"
CID.SvcDesc="Controller Interface Device"
```

The [*Services*] and [*AddService*] sections informs the PnP manager to load the appropriate files. A key will be created in HKEY_LOCAL_LOCAL_MACHINE \System\CurrentControlSet\Services in the registry. The *ServiceType=1* value indicates that the service entry is a kernel-mode driver and needs to be loaded while booting Windows 2000. The *ErrorControl* value indicates that in case of an error, the system must log the error but this must not halt the system from booting up. The *ServiceBinary* line gives the name of the driver that must be stored in the registry. The *AddReg* section adds entries to the Windows system registry. The HKR key specifies the relevant registry entry [10], which is the CIDII.sys. The default installer adds the DevLoader and NTMPDriver values as listed in DDK. The [*Strings*] section is usually used to describe the manufacturer details and the hardware version information.

6.3 An Easy Way to Write INF Files

There are various tools available that automate the process of writing INF files for the hardware devices. Microsoft provides *infedit, Geninf, ChkINF* tools to create INF files. These tools obtain the information required

by asking the user different questions about the hardware device. This chapter equips the user with the information needed to install the device driver. The user has the option of writing his/her own INF file or using the automated tools provided. The Microsoft DDK provides a wealth of information about writing INF files.

Chapter 7 Performance Results from Traffic Measured on the USB Bus

When the communication latencies are in milliseconds, it is pretty difficult to measure the speed. A specialized tool called the USB Bus Analyzer, shown in Figure 7.1, was used for this project. This device is capable of logging all the packets on the USB bus.

Figure 7.1: CATC Advisor and Protocol Analyzer.

This analyzer uses hardware triggering to capture all of the real-time events on the USB bus. All the recorded events can be viewed in the software provided with the package. There are options to configure the software of the analyzer. When using this analyzer, it is important to set the option to record at classic USB speeds or USB 1.1 speeds.

7.1 CATC trace of the driver performance

In order to compare the latency of the old driver, the CATC trace was measured for a read and a write operation. A series of *reads* from the CID recorded is shown in Figure 7.2. The CATC trace in Figure 7.2 shows 6 *Start-of-Frames (SOF)* between the two reads. Therefore we have a delay of 6 ms between two consecutive reads. The write operation also exhibits a similar delay of 6 ms between the two write operations in Figure 7.3. Both results have been consistent over a large number of tests.

With the new driver installed on the target machine, a similar Win32 application is written to verify the speed of the driver. The Win32 application reads data repeatedly from the CID II and the trace is shown in Figure 7.4. Here in the screenshot, there is a read at every SOF. A SOF in USB is every 1 millisecond; hence we can see that the data is being read every 1 millisecond.

The Win32 application is also programmed to write a series of 73 bytes to the CID. A similar trace was recorded for the writing 73 bytes to the CID shown in Figure 7.5. The CATC trace also has other various useful information. These features help the designers troubleshoot various problems. The data block shows 73 bytes

for write and 9 bytes for a read operation. If the display options in the CATC software are changed, then this data block can provide the contents of the data block.

All the packets of USB bus are displayed with numbers in the trace. These numbers are useful to calculate the time between any two packets. The frame numbers provided can be used to verify if the transfers are occurring at the correct frame, if that option is specified in the device driver. The direction of the transfer is also indicated with an arrow, in addition to the *IN* and *OUT* markers.

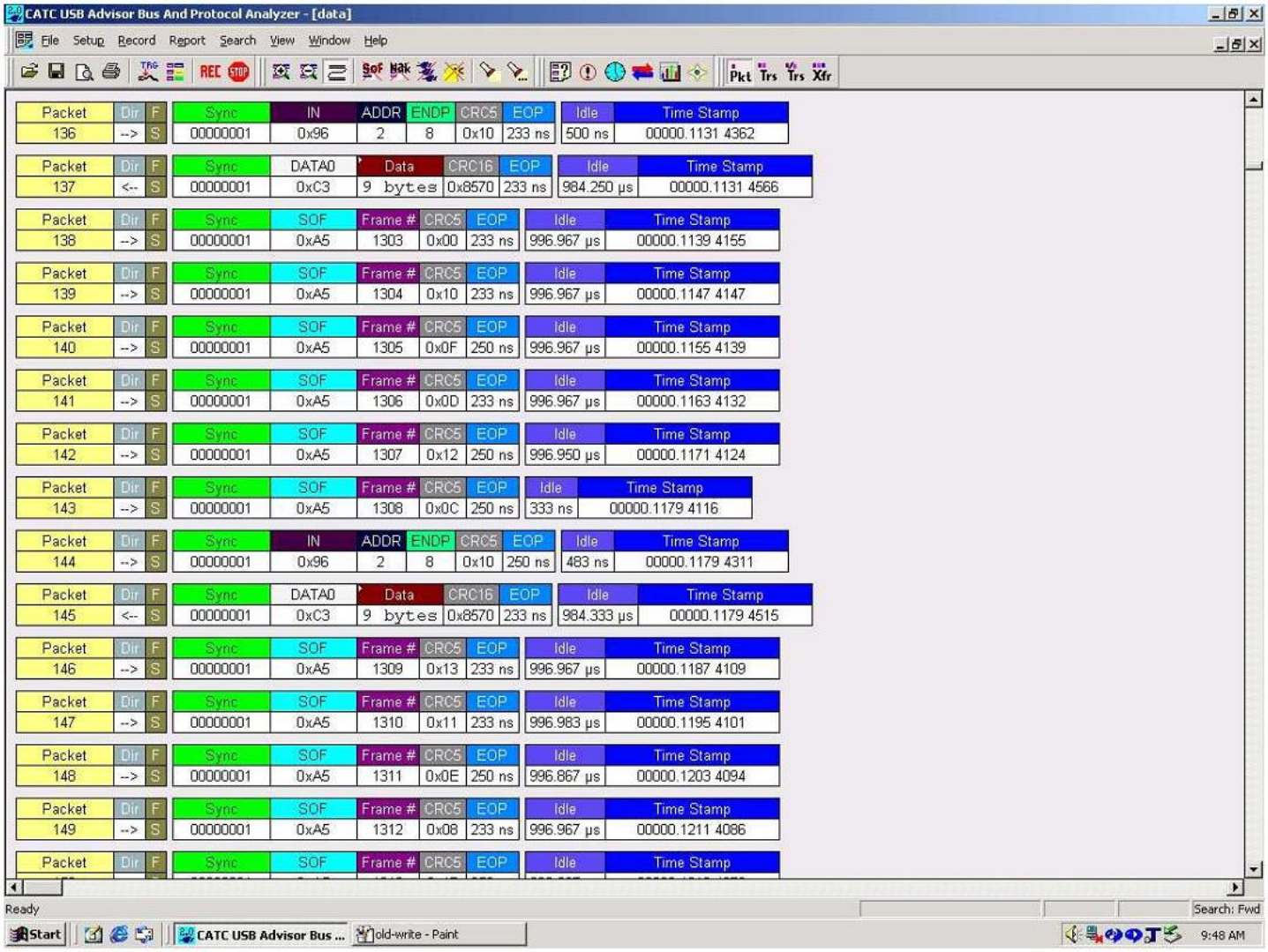

Figure 7.2: Screenshot of the CATC trace for read operations using the old general purpose driver.

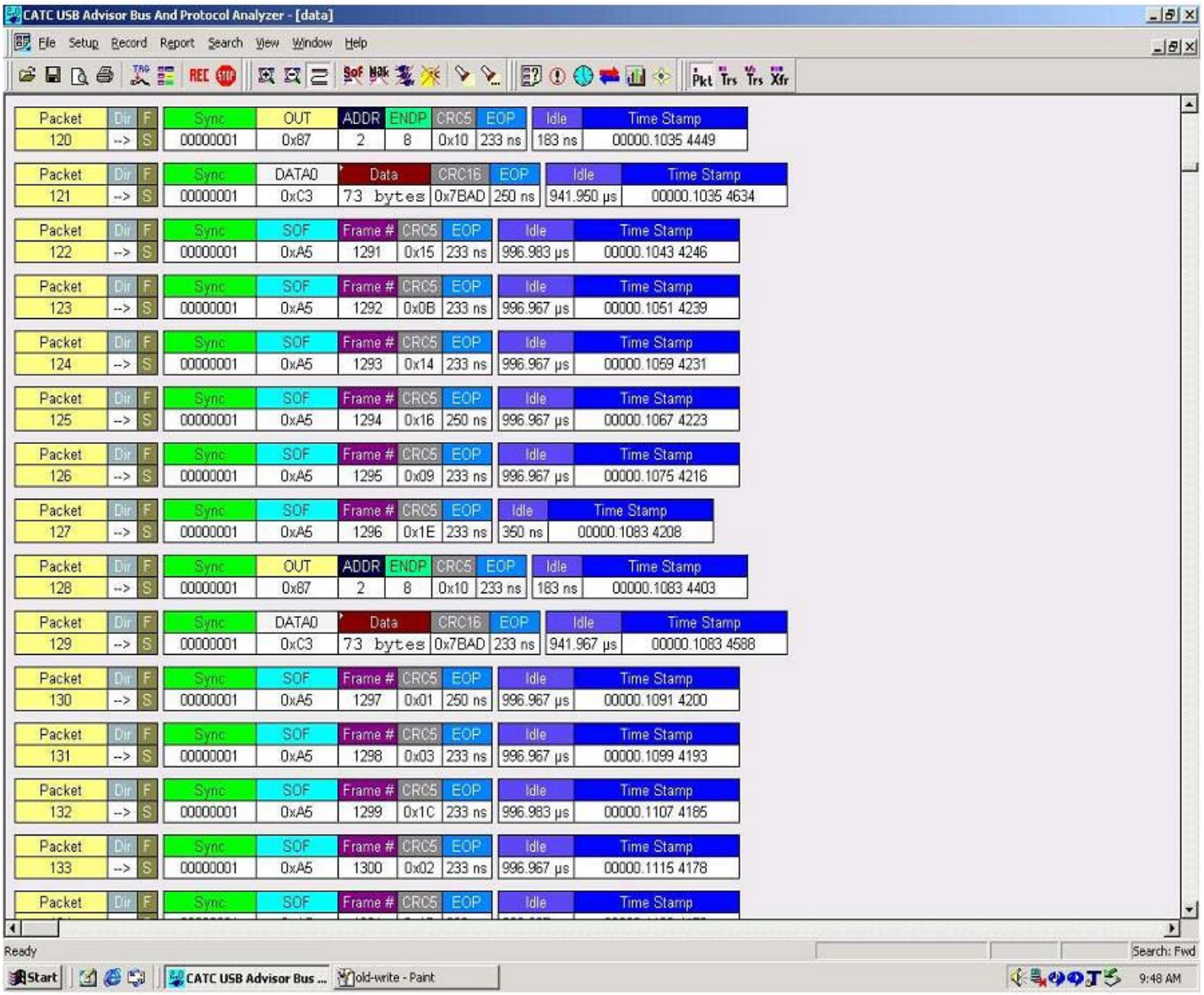

Figure 7.3: Screenshot of the CATC trace for write operations using the old general purpose driver.

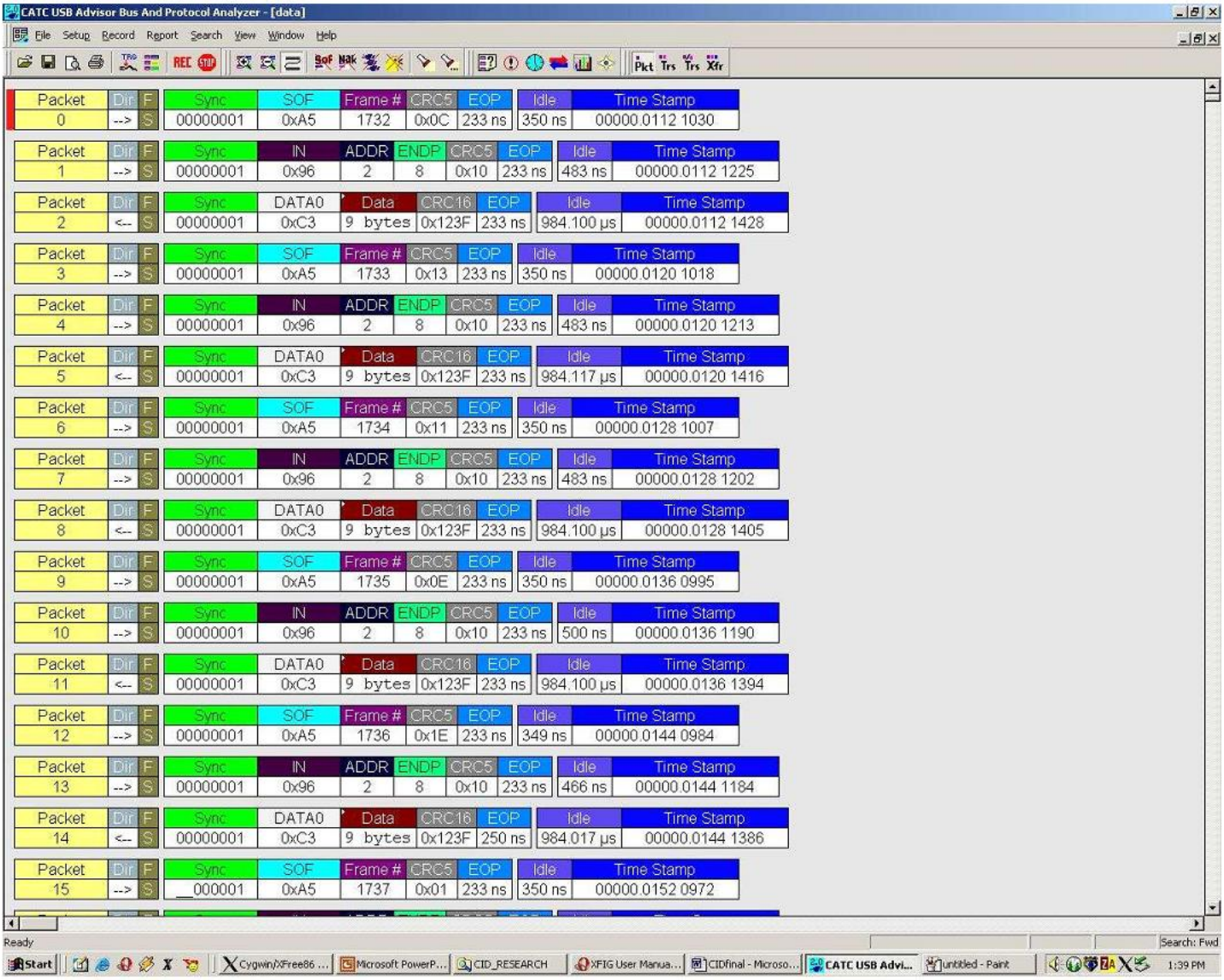

Figure 7.4: Screenshot of the CATC trace for read operations using new CID II driver.

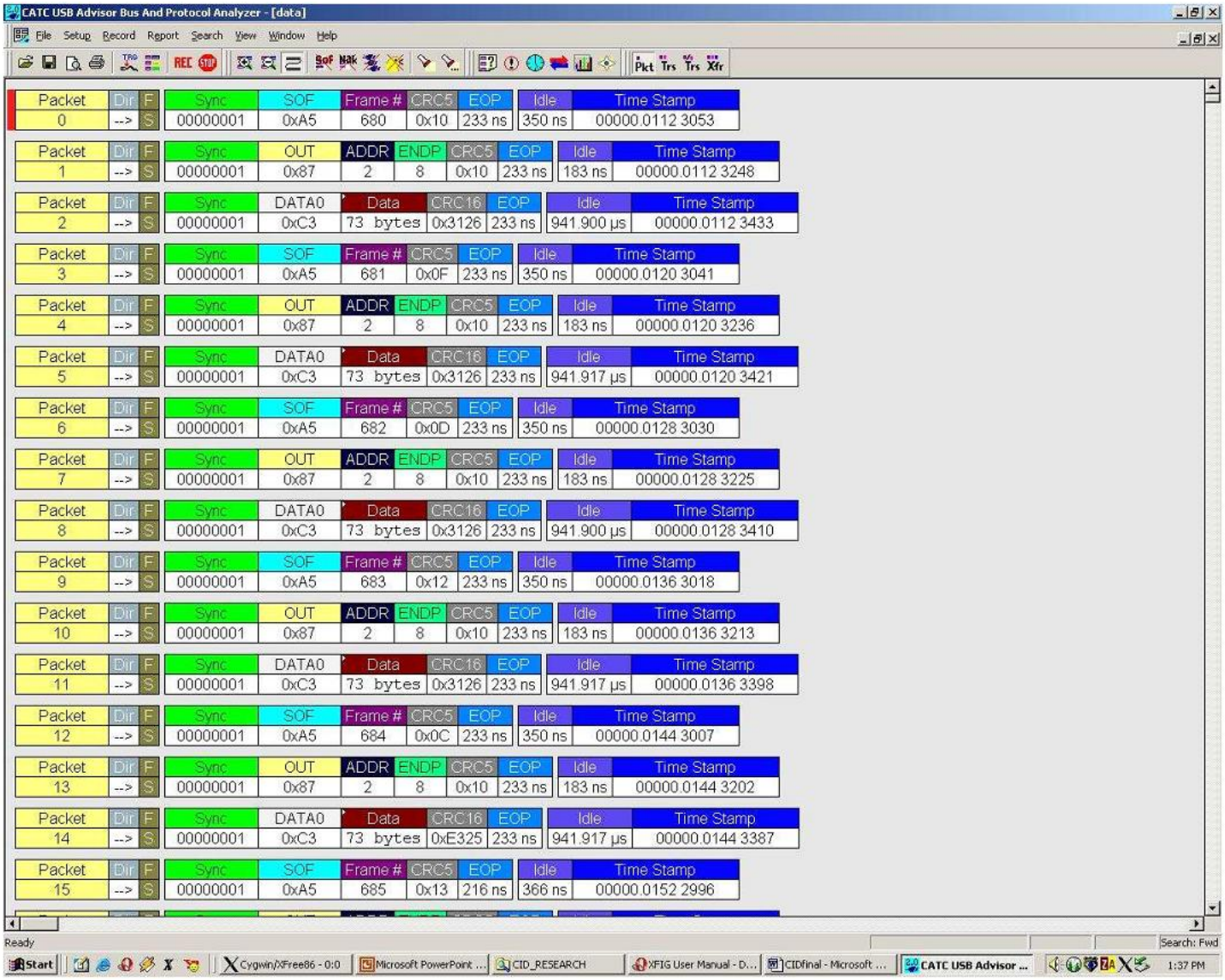
CATC USB Advisor Bus And Protocol Analyzer - [data]
File Setup Record Report Search View Window Help
Packet Dir F Sync SOF Frame # CRC5 EOP Idle Time Stamp
0 --> S 00000001 0xA5 680 0x10 233 ns 350 ns 00000.0112 3053
Packet Dir F Sync OUT ADDR ENDP CRC5 EOP Idle Time Stamp
1 --> S 00000001 0x87 2 8 0x10 233 ns 183 ns 00000.0112 3248
Packet Dir F Sync DATA0 Data CRC16 EOP Idle Time Stamp
2 --> S 00000001 0xC3 73 bytes 0x3126 233 ns 941.900 µs 00000.0112 3433
Packet Dir F Sync SOF Frame # CRC5 EOP Idle Time Stamp
3 --> S 00000001 0xA5 681 0x0F 233 ns 350 ns 00000.0120 3041
Packet Dir F Sync OUT ADDR ENDP CRC5 EOP Idle Time Stamp
4 --> S 00000001 0x87 2 8 0x10 233 ns 183 ns 00000.0120 3236
Packet Dir F Sync DATA0 Data CRC16 EOP Idle Time Stamp
5 --> S 00000001 0xC3 73 bytes 0x3126 233 ns 941.917 µs 00000.0120 3421
Packet Dir F Sync SOF Frame # CRC5 EOP Idle Time Stamp
6 --> S 00000001 0xA5 682 0x0D 233 ns 350 ns 00000.0128 3030
Packet Dir F Sync OUT ADDR ENDP CRC5 EOP Idle Time Stamp
7 --> S 00000001 0x87 2 8 0x10 233 ns 183 ns 00000.0128 3225
Packet Dir F Sync DATA0 Data CRC16 EOP Idle Time Stamp
8 --> S 00000001 0xC3 73 bytes 0x3126 233 ns 941.900 µs 00000.0128 3410
Packet Dir F Sync SOF Frame # CRC5 EOP Idle Time Stamp
9 --> S 00000001 0xA5 683 0x12 233 ns 350 ns 00000.0136 3018
Packet Dir F Sync OUT ADDR ENDP CRC5 EOP Idle Time Stamp
10 --> S 00000001 0x87 2 8 0x10 233 ns 183 ns 00000.0136 3213
Packet Dir F Sync DATA0 Data CRC16 EOP Idle Time Stamp
11 --> S 00000001 0xC3 73 bytes 0x3126 233 ns 941.917 µs 00000.0136 3398
Packet Dir F Sync SOF Frame # CRC5 EOP Idle Time Stamp
12 --> S 00000001 0xA5 684 0x0C 233 ns 350 ns 00000.0144 3007
Packet Dir F Sync OUT ADDR ENDP CRC5 EOP Idle Time Stamp
13 --> S 00000001 0x87 2 8 0x10 233 ns 183 ns 00000.0144 3202
Packet Dir F Sync DATA0 Data CRC16 EOP Idle Time Stamp
14 --> S 00000001 0xC3 73 bytes 0xE325 233 ns 941.917 µs 00000.0144 3387
Packet Dir F Sync SOF Frame # CRC5 EOP Idle Time Stamp
15 --> S 00000001 0xA5 685 0x13 216 ns 366 ns 00000.0152 2996
Ready
Search: Fwd
Start
Cygwin/XFree86 - 0:0
Microsoft PowerPoint ...
CID_RESEARCH
XFIG User Manual - D...
CIDfinal - Microsoft ...
CATC USB Advisor ...
1:37 PM

Figure 7.5: Screenshot of the CATC trace for write operations using the new CID II driver.

7.2 Driver Performance with Multiple Controller Interface Devices

The driver sends and receives data every millisecond for each CID II when there is more than one on the USB bus. However when a switch occurs between the CIDs, there is still a delay of 6 ms. This delay is attributed to the bus polling done by the USB host to locate any other new devices attached to the USB bus. Chapter 8 outlines some of the techniques that could possibly eliminate this delay. The new USB driver improved the performance of the CID II, but there is still some room for improvement. However, the improvements presented here will allow a few more CIDs to be connected in one simulation when time steps of 100 ms are used.

Table 7.1: Comparison of General Purpose Driver and New Driver with Multiple CID Configurations

Device Drivers	READ operation in multiple CIDS			
	CID 1	Switch time	CID 2	Total Latency
Old general purpose driver	6 ms	6 ms	6 ms	6 ms
New CID II driver	1 ms	6 ms	1 ms	8 ms
	WRITE operation in multiple CIDS			
	CID 1	Total Latency	CID 2	Total Latency
Old General Purpose Driver	6 ms	6 ms	6 ms	18 ms
New CID II driver	1 ms	6 ms	1 ms	8 ms

Chapter 8 Summary and Future Work

8.1 Summary

A device driver for CID II was designed, coded, and tested for performing isochronous mode of transfers. This device driver is system software that enables the computer to recognize the CID II when it is connected. Drivers for the most commonly used hardware devices are provided with the operating system, but special purpose device drivers for new hardware devices are normally provided by the hardware vendors. Some of the drivers are designed as general purpose drivers and they are not always efficient for custom hardware.

A custom device driver can provide more control and efficiency for the hardware device. This thesis discussed the design, coding, and testing of such a custom device driver for the CID II. This device driver designed for the CID II eliminates the latency of 6 milliseconds involved in a read or write operations with one CID. This driver was successfully tested for its performance. The driver performed equally well on both slower and faster computers.

8.2 Suggested Future Work

Driver development is a recursive process, in which various additions and deletions in code are made to reflect the changes introduced in the hardware. Some of the interesting future research directions are:

(1) The CID driver is mainly designed for the isochronous mode of transfers. Though isochronous mode transfers guarantees bounded transfer latency, there is also a possibility of using bulk transfers for the CID II. Research options using bulk transfers are currently being evaluated. If the bulk transfer method is implemented for the CID II, then the device driver also needs to be modified.

(2) The current device driver has reduced the delay when communicating with one CID II. But there is a delay involved when multiple CID II are connected. Reduction of this delay should be the priority. This delay is caused when the control from the driver returns to the test program to obtain the handle for the next CID. There are various possibilities of eliminating this delay, two recommendations have been made:

(a) Obtain all the CID handles: The control switches back to the main program to obtain the handle for the new CID to perform the transfer rather than polling the bus. This switching could then happen in a faster, predefined sequence. Use of multiple threads would be a good direction towards solving the problem of communicating with 40 CIDs in 1 single frame, since threads have fast context switching.

(b) Using Asynchronous Transfers: This solution was suggested by Troy Cuff, when he was an undergraduate student in Electrical and Computer Engineering at the University of Idaho. This method would employ asynchronous transfers, where the driver would send in requests to multiple CIDs at the same time. This method requires certain modification in the driver code. An abstract representation of the idea is described in the Figure 8.1.

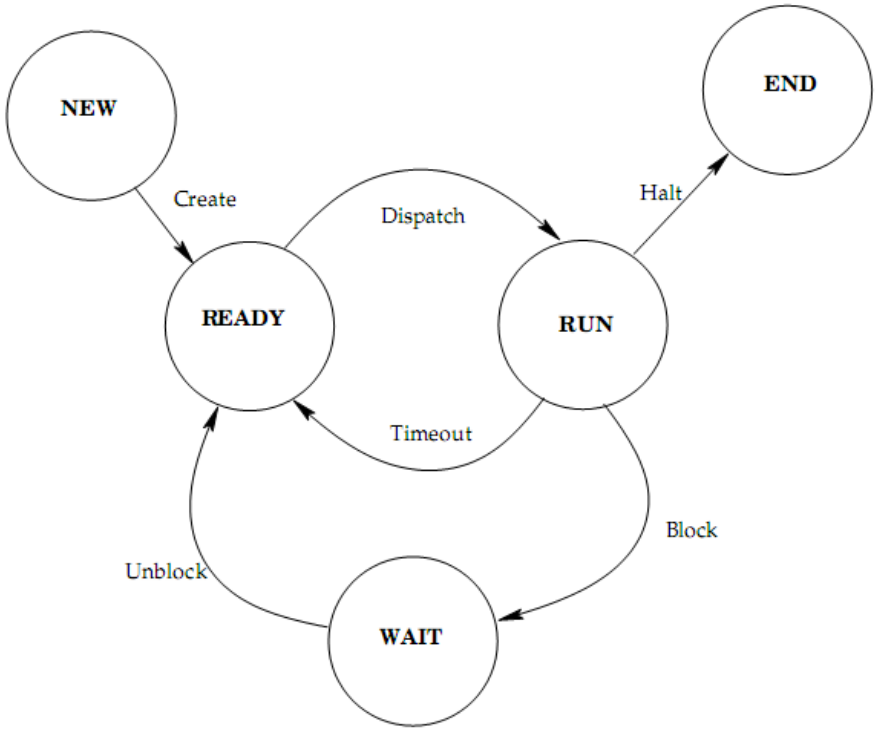

Figure 8.1: Asynchronous transfers.

(3) Research can also be done to test different CID.

(a) One interesting test would be to connect 20 or more CIDs and to measure the latency involved to see if it scales linearly with number of devices.

(b) A second method would be to turn off the hardware acceleration on the computer and to measure the performance. There is research being done on impact of graphics acceleration on isochronous modes in USB devices.

References

[1] Engelbrecht, R.J., K.N. Balke, S.P. Venglar, S.R. Sunkari., *Recent Applica tions of Hardware-in-the-Loop Traffic Simulation*, Compendium of Papers of the 70th Annual Meeting of the Institute of Transportation Engineers (CD-ROM), Institute of Transportation Engineers, Washington D.C., 2000.

[2] Federal Highway Administration, *Traffic Software Integrated System 97 User's Guide - CORSIM Version 1.03*, New York June 1997.

[3] Compaq, Intel, Microsoft, NEC, *Universal Serial Bus Specification*, Revision 1.1 September 23, 1998.

[4] Ying Zhou, *The Design of the NIATT Controller Interface Device*, M.S Thesis, University of Idaho, May 2000.

[5] Zhen Li, *Assessing the Feasibility of Using Hardware-in-the-Loop Real Time Simulation to Improve the Signal Timing Design Process*, Ph.D Thesis, University of Idaho, December 2003.

[6] John Fisher, *A Method of Verifying the Accuracy of Real-Time Hardware-in-the-loop Traffic Simulation*, M.S Thesis, University of Idaho, July 2002.

[7] Bullock, D., and A. Catarella., *A Real-Time Simulation Environment for Evaluating Traffic Signal Systems*, Paper presented at the 77th Annual Transporation Research Board Meeting, Washington D.C., January 1998.

[8] Walter Oney, *Programming the Microsoft Windows Driver Model*, Microsoft Press, Redmond, Washington 1999.

[9] Jan Axelson, *USB Complete, Penram International Publishing*, India 2001.

[10] Chris Cant, *Writing Windows WDM Device Drivers*, Research and Development Books, Lawrence KS, 1999.

[11] *Microsoft Windows 2000* DDK, Driver Writer's Guide, Chapter 2 "Building Drivers."

[12] *Innovative Transportation Concepts Inc*, ITC. (2003), 1128 NE 2nd St., Ste. 204 Corvallis, OR 97330.

Appendix A CID II Settings

This section describes how the 6 ms latency for a read/write cycle in the USB device driver was determined. This test was done by John Fisher's for his thesis "A Method of Verifying the Accuracy of Real-Time Hardware-in-the-loop Traffic Simulation." The steps to determine the delay were:

(1) First, the CID's firmware was modified to pulse a microcontroller output pin whenever a command was received from the computer.

```
void SOF_Interrupt(void)
{
    if (OUT8DATA)
      SET_DB1();
    CLR_DB1();
}
```

(2) The computer was then programmed to write to the CID repeatedly.

```
while (1)
    cid.Write(1, buffer);
```

(3) An oscilloscope was used to monitor the microcontroller pin being toggled. There were pulses occurring consistently every 6 ms: the computer-to-CID write time.

(4) A similar test to measure the latency from the CID to the computer (a "read" operation) was done as follows:

```
unsigned long int lastClock;
cid.SetMode(4);
while (1)
{
   cid.ReadQueueStatus(); // read clock, among other things
   printf("%i ", cid.clk - lastClock);
   lastClock = cid.clk;
}
```

This also revealed a consistent "read" latency of 6 ms.

A.2 IOCTL Definitions

```
//////////////////////////////////////////////////////
//
// IOCTL Definitions
//////////////////////////////////////////////////////

#define Isousb_IOCTL_INDEX 0x0000

#define IOCTL_Cid_GET_DEVICE_DESCRIPTOR CTL_CODE(
FILE_DEVICE_UNKNOWN, \
           Isousb_IOCTL_INDEX+0,\
           METHOD_BUFFERED, \
           FILE_ANY_ACCESS)

#define IOCTL_Cid_GET_CONFIGURATION_DESCRIPTOR CTL_CODE(
                      FILE_DEVICE_UNKNOWN, \
                      Isousb_IOCTL_INDEX+1,\
                      METHOD_BUFFERED, \
                      FILE_ANY_ACCESS)

#define IOCTL_Cid_RESETPIPE CTL_CODE(FILE_DEVICE_UNKNOWN, \
                      Isousb_IOCTL_INDEX+2,\
                      METHOD_IN_DIRECT, \
                      FILE_ANY_ACCESS)

#define IOCTL_Cid_ABORTPIPE CTL_CODE(FILE_DEVICE_UNKNOWN, \
                      Isousb_IOCTL_INDEX+3,\
                      METHOD_IN_DIRECT, \
                      FILE_ANY_ACCESS)
```

```
#define IOCTL_CID_ISO_WRITE CTL_CODE(FILE_DEVICE_UNKNOWN, \
                      Isousb_IOCTL_INDEX+4,\
                      METHOD_IN_DIRECT, \
                      FILE_ANY_ACCESS)
```

A.3 Debugging settings on host machine

```
//////////////////////////////
////////boot.ini file///////
//////////////////////////////
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Microsoft
Windows 2000 Professional" /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Microsoft
Windows 2000 Professional" /fastdetect /debug
/debugport=com1 /baudrate=57600
```

A.4 Path settings to load symbols

```
A sample batch file before starting windbg.exe on host machine
//////////////////////////////
/////debugger_setup.bat/////
//////////////////////////////
set _NT_SYMBOL_PATH=c:\winnt\symbols
set _NT_DEBUG_PORT=com1
set _NT_DEBUG_BAUD_RATE=57600
set _NT_LOG_FILE_OPEN=c:\DEBUG.LOG
```

CID Design Team

Management

Brian K. Johnson, Principle Investigator

Richard W. Wall, Co-Principle Investigator

Technical Advisors

Darcy Bullock, Consultant (Purdue University)

Mike Kyte, NIATT Director

Peter Kohl and Peter Ragsdale, McCain Traffic Supply

Ron Nelson, MRCI Staff Engineering (hardware trouble-shooting)

Graduate Research Assistants

Zhen Li, Ph.D. student, Civil Engineering

Eugene Bordenkircher, MS student, Computer Engineering

Manjunatha Reddy, MS Student, Computer Engineering

Undergraduate Project Assistants

Troy Cuff, Computer Engineering

Eresh Suwal, Computer Engineering (NIATT Intern)

Tyrel Jensen, Electrical Engineering (NIATT Intern)

Zeljko Mijatovic, Electrical Engineering (NIATT Intern)

Henis Mitro, Electrical Engineering (NIATT Intern)

Sam Young, Electrical Engineering (NIATT Intern)